TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ARTTU LEPPÄKOSKI

LIFE CYCLE MANAGEMENT FOR PROGRAMMABLE MACHINE CONTROL PLATFORM

Master's Thesis

Examiner: Professor Timo D. Hämäläinen
Examiner and topic approved by the Faculty Council of the Faculty of Engineering Sciences on 6 February 2013.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
LEPPÄKOSKI, ARTTU: Ohjelmoitavan koneenohjausalustan elinkaarenhallinta
Master's Thesis, 78 sivua, 4 liitesivua
Maaliskuu 2013
Pääaine: Ohjelmoitavat alustat ja laitteet
Tarkastaja: professori Timo D. Hämäläinen
Avainsanat: ohjelmoitava koneenohjausalusta, sulautetun ohjelmiston kehitysympäristö, SCM, Yocto Project

Tässä työssä tutkittiin ohjelmoitavan koneenohjausalustan elinkaarenhallintaa laitteiston ja ohjelmiston näkökulmasta. Elinkaarenhallinta on erityisen tärkeää, sillä alustan elinikä on viidestä kahteenkymmeneen vuotta. Huomioon otettavia näkökohtia ovat ohjelmiston uudelleentuotettavuus, ohjelmistotuotteiden hallinta, versionhallinta, testiautomaatio, jatkuva integrointi sekä ohjelmiston konfiguroitavuus. Näitä hallitaan prosesseilla, joita ovat ohjelmistokonfiguraatioiden hallinta, sovelluksen elinkaarenhallinta sekä tuotteen elinkaarenhallinta. Nämä prosessit on esitelty tässä työssä ohjelmoitavan koneenohjausalustan näkökulmasta.

Työn ytimenä on sulautetun ohjelmiston kehitysympäristö, jolla luodaan järjestelmän käyttämä ohjelmistoarkkitehtuuri. Aiemmin käytössä ollut sulautetun ohjelmiston kehitysympäristö ei soveltunut monen eri tuotteen hallintaan eikä tukenut kattavasti eri laitealustoja. Käytössä olleelle kehitysympäristölle ei myöskään ollut tarjolla kattavaa tukea sekä sen käytettävyys oli huonoa. Tässä työssä otettiin käyttöön uusi kehitysympäristö. Työssä vertailtiin avoimia ja kaupallisia vaihtoehtoja sulautetun järjestelmän kehitysympäristöksi. Työn toteutukseen valittiin Yocto Project, jonka todettiin soveltuvan parhaiten käytössä olevan ohjelmoitavan koneenohjausalustan ohjelmistokonfiguraatioiden hallintaan.

Työssä tutkittiin ja määriteltiin menetelmiä, joilla alustan elinkaarenhallinta voidaan toteuttaa tehokkaasti ja yksinkertaisesti käyttäen Yocto Projectia. Nämä menetelmät käsittivät ohjelmiston jakamisen eri konfiguraatioihin ja versioihin, ohjelmiston versionhallinnan, uudelleentuotettavuuden toteuttamisen sekä sovellusten luomiseen käytettävän prosessin.

Työssä luotiin ohjelmoitavalle koneenohjausalustalle yksilöllinen Yocto Project -kerros, joka mahdollistaa alustan ohjelmiston ja myös laitteiston muutokset vaivatta. Kerroksen luomisen yhteydessä luotiin kaikkiaan 76 erilaista tiedostoa, jotka sisältävät yhteensä yli 3000 koodiriviä. Yhdessä kehitysympäristön kanssa käyttöön otettiin elinkaarenhallintaa tukevia työkaluja ja ohjelmistoja, kuten palvelin jatkuvaa integrointia ja testiautomaatiota varten. Kerroksen toteutukseen ja työkalujen käyttöönottoon kului yhteensä 180 työtuntia. Näiden mitattujen arvojen pohjalta muodostettiin arvioita eri työtehtävien suorittamiseen tarvittavalle ajalle. Työssä luotiin myös mittareita, joiden avulla voidaan seurata kehitysprosessin ja työkalujen toimivuutta.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Automation Technology
LEPPÄKOSKI, ARTTU: Life Cycle Management for Programmable Machine Control Platform
Master of Science Thesis, 78 pages, 4 Appendix pages
March 2013
Major: Programmable platforms and devices
Examiner: Professor Timo D. Hämäläinen
Keywords: Programmable machine control platform, embedded software framework, SCM, Yocto Project

This Thesis develops methods for managing software and hardware during the life cycle of a programmable machine control platform. Life cycle management is important because of the long life cycle of this platform. Software reproducibility, management of software products, version controlling, test automation, continuous integration, and configurability of software are considered. These issues are implemented by processes for software configuration management, application life cycle management, and product life cycle management. The processes are described within the contexts of a programmable machine control platform in this Thesis.

The core of this work is embedded software framework, which creates the software architecture for the platform. A previously used embedded software framework was not suitable for controlling multiple software products and it only supported few hardware platforms. In addition, the lack of community and enterprise support and the framework's poor usability were major disadvantages. Therefore, a new embedded software framework was deployed in this Thesis. Open-source and commercial options for an embedded software framework were compared and the Yocto Project was chosen. It was the best option meeting the requirements on programmable machine control platform. Based on it, new methods were studied and specified for managing software configurations and versions, reproducibility, version controlling, and application creation.

A new layer for the Yocto Project was also created in this Thesis. The new layer simplifies software and hardware modifications significantly. The layer consists of 76 files, including over 3000 lines of code and configuration information. In addition, a new server for continuous integration and test automation was deployed. It took a total of 180 working hours to create the layer and deploy the additional tools. Measurements were gathered to estimate the workload for different tasks. Furthermore, a number of metrics were created to provide a simple way to estimate the effectiveness of the tools and the processes.

PREFACE

This work was carried out at Konecranes, Hyvinkää in autumn 2012 and winter 2013.

Assistance and support from many people have made this Thesis possible. I would like to thank Timo Sorsa for this opportunity to write this Thesis at Konecranes. I would also like to thank my supervisor Jouni Jocklin for the guidance during this project.

I want to thank Professor Timo D. Hämäläinen for the great guidance and valuable comments during this work.

This Thesis would not have been possible without the support from my family during my studies.

Finally, I would like to thank everyone who has been helping me during this project.

Arttu Leppäkoski

Hyvinkää, 6.3.2013

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ADT	Application Development Toolkit
ALM	Application Life cycle Management
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Advanced RISC Machines
BSP	Board Support Package
CCB	Change Control Board
CI	Configuration Item
CM	Configuration Management
DSP	Digital Signal Processor
ERP	Enterprise Resource Planning
FTP	File Transfer Protocol
GNU	GNU's Not Unix
GPL	GNU General Public License
HAL	Hardware Abstraction Layer
HdS	Hardware-dependent Software
HW	Hardware
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
ISO	International Organization for Standardization
JTAG	Joint Test Action Group
LTIB	Linux Target Image Builder
LTS	Long-Term Support
MIPS	Microprocessor without Interlocked Pipeline Stages
OS	Operating System
PBD	Platform-Based Design
PC	Personal Computer
PCB	Printed Circuit Board
PCCP	Programmable Crane Control Platform
PDK	Product Development Kit
PLC	Programmable Logic Controller
PLM	Product Life cycle Management
PMCP	Programmable Machine Control Platform
R&D	Research & Development
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory

RTC	Real-Time Clock
RTOS	Real-Time Operating System
SCM	Software Configuration Management
SD	Secure Digital
SDLC	Systems Development Life Cycle
SoC	System on a Chip
SPI	Serial Peripheral Interface
SVN	Subversion
SW	Software
TUT	Tampere University of Technology
USB	Universal Serial Bus
VCS	Version Control System
WLAN	Wireless Local Area Network

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Background

Advances in manufacturing of electronic devices and reductions in manufacturing costs have enabled embedded systems to become more popular. During the last decades, embedded systems have become larger and more powerful but at the same time, the complexity of software has increased exponentially. This has led to the situation where size of the software has grown faster than the productivity of the software development. In a modern world, products should be in the market as soon as possible. Development time should be as short as possible but at the same time, quality of the product should remain high. Common problems with the system development are poor product quality, low development productivity, and long development cycle time (TUT, 2012). In the case of embedded system design, this means that hardware should be properly functioning and device tested thoroughly.

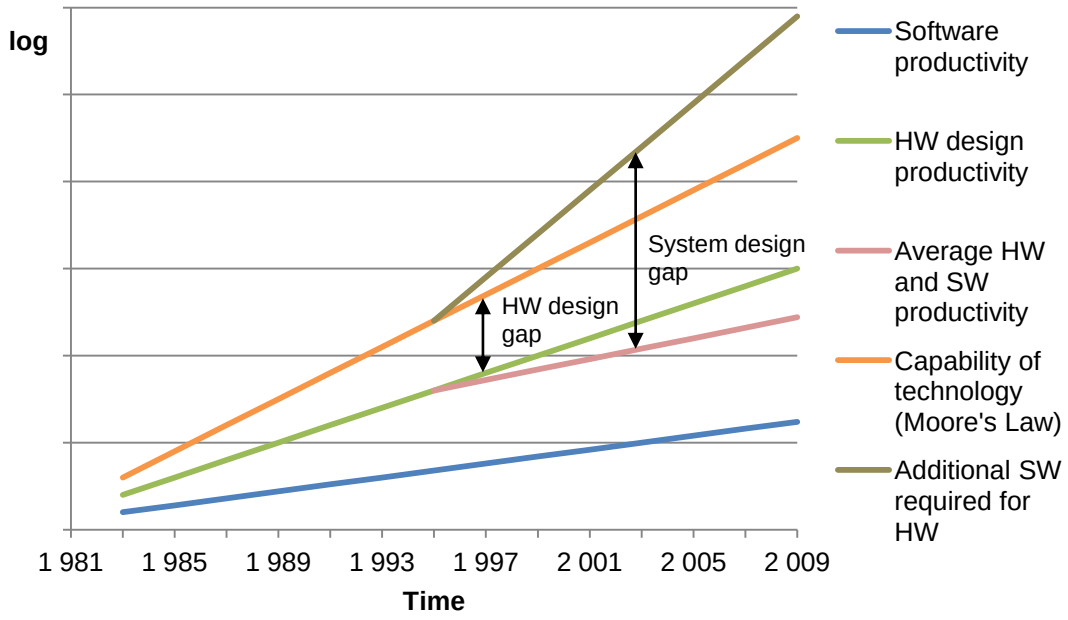

Figure 1.1: Hardware and software productivity gaps (adapted from (Ecker, et al., 2009))

Figure 1.1 illustrates that the design productivity gap still keeps growing (Dömer, et al., 2009). System design productivity gap illustrates the difference between the capabilities of the current technologies and productivity. Software productivity doubles every five years. However, capability of the technology doubles every 18 months and the hardware design productivity almost doubles every 18 months. This evolution has led into the

situation where all potential resources of the hardware becomes harder and harder to be utilized by the software. (Dömer, et al., 2009)

Traditionally, specific functions of an industrial machine are implemented using several independent devices realizing the functions. These devices could be WLAN-module, programmable logic controller and Ethernet switch. *Programmable machine control platform (PMCP)* integrates all these functions into one device, allowing it to be updated and modified afterwards. PMCP is used to control and monitor industrial machines. Industrial environment calls for special requirements for safety, quality, and performance.

In the machine industry, life cycle length for the PMCP is from five to 20 years. The long life cycle sets requirements for software and hardware. The PMCP should be upgradable, robust, and easy to maintain. At the same time, all process steps that have been carried out in the past should be repeatable, reproducible and traceable. It is also possible that problems with component availability will require components to be replaced. Issues described require attention to the implementation and well organized life cycle management for the products.

In the case of system development, almost 80% of the system content and functionality is implemented in the software (Dömer, et al., 2009). Large amount of the costs during the software's life cycle is caused by the maintenance operations. Life cycle costs of the software are illustrated in Figure 1.2. Management of the life cycle and software configurations becomes important with the PMCP because of the long life cycle, domination of the software, increasing design productivity gap, and work needed for the maintenance.

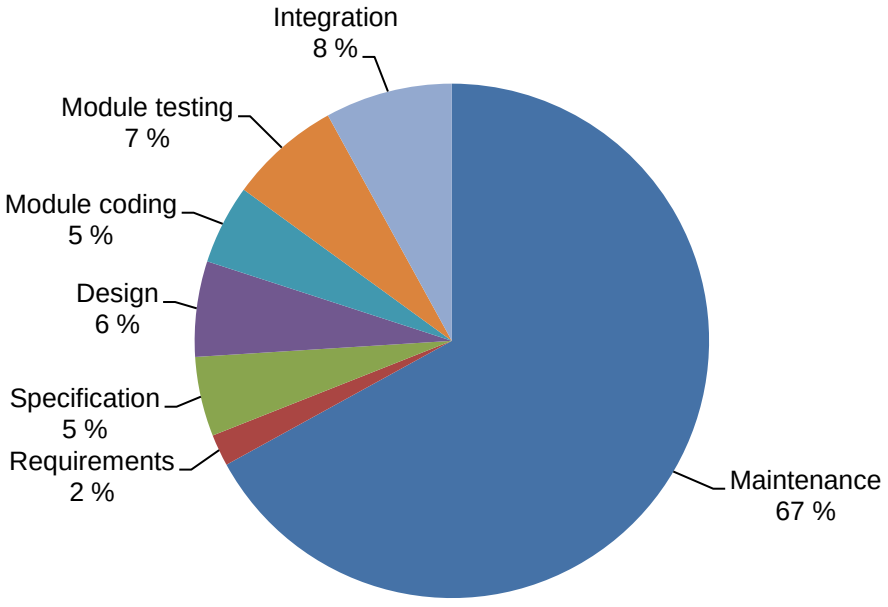

Figure 1.2: Approximated life cycle costs of software (Schach, 2010)

As Leon (2005) argues, usage of the scientific methods with the software (and system) development provides several benefits. These benefits allow personnel of the company to be more receptive for new methods, procedures, technologies, and ideas and to be more productive.

This Thesis is based on an existing prototype implemented to test capabilities of the PMCP for industrial crane controlling (Jocklin, 2011). In this Thesis, this prototype is referred as a *programmable crane control platform (PCCP)*. Theory introduced in this Thesis is not only applicable for this specific prototype; instead, this Thesis discusses issues involved with the PMCP in general. However, practical part of this Thesis is related to the PCCP. The main reason for this Thesis is to study how technical aspects of the life cycle management and software configuration management should be utilized with the PMCP.

1.2 Objectives

Currently used methods with the PCCP are considered troubled and clumsy. In addition, there are no well-defined guidelines for the life cycle management. Therefore, this Thesis develops methods and guidelines for the life cycle management and software configuration management with the PCCP. Main objectives for this Thesis are following:

- Introduce and define product and application life cycle management
- Introduce and define software configuration management
- Study and compare embedded software frameworks and evaluate the relationship between the life cycle and software configuration management and embedded software framework
- Implement a case with one selected embedded software framework
- Determine development guidelines

1.3 Structure of the Thesis

This Thesis includes parts for both theoretical and practical issues.

Chapter 2 introduces the basics for the PMCP.

Chapter 3 describes basic models and activities used with the software and system development. Life cycle management methods are also introduced in this Chapter.

Chapter 4 introduces the software configuration management and activities, phases, and tools used with it. It is also described how the software configuration management should be taken into account with the PMCP.

Chapter 5 introduces the concept of an embedded software framework. Features, methods, and usage of the embedded software frameworks are also described in this Chapter. Furthermore, the most important open source and commercial embedded software frameworks are introduced and compared. Usage of the embedded software framework for the software configuration management is also described in this Chapter.

Chapter 6 introduces and describes methods, processes, and guidelines to realize the life cycle management for the PCCP. These are described within the context of technical side.

Chapter 7 contains a description of a case to deploy an embedded software framework for the PCCP.

Chapter 8 discusses about the results of the work and describes what should be done in the future.

2 PROGRAMMABLE MACHINE CONTROL PLATFORM

2.1 Introduction

PMCP is designed to control specific functions or as a general platform that is customized. This customization is realized by adding or removing hardware modules or implementing specific software. In this Thesis, PMCP is assumed to be customizable. The PMCP follows the Platform-Based Design (PBD) that is an approach where integrated and verified platforms serve as a basis for families of derivative products (Bailey, et al., 2005). Main benefits for the PMCP are cost-efficiency, high configurability, and reliability (due to careful testing).

PMCP differentiates from a desktop system by being cost-sensitive and having real-time and power constraints. PMCP can be implemented by using a variety of processors and processor architectures, but also with fewer system resources than desktop system. In addition, PMCP may be operating under extreme environmental conditions and software failures are usually much more severe than with the desktop systems. All of the issues described above emphasize the importance of using efficient methods and tools throughout the design process and the life cycle of the PMCP. (Berger, 2002)

Architecture of the PMCP consists of three main layers described in Figure 2.1. Layers are hardware platform, software platform, and applications. These layers are described in the following Sections from bottom up (from the physical hardware to the applications).

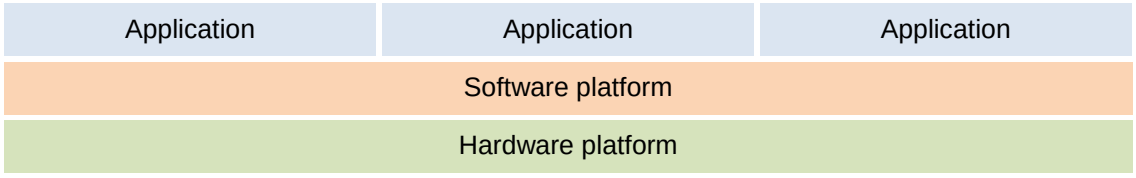

Figure 2.1: Layers of the PMCP

2.2 Hardware platform

Hardware platform consists of physical hardware components and interconnections and it can be assembled in several ways. First practice is to place all components to a single Printed Circuit Board (PCB). This approach does not provide room for customization but it is simple and cost-efficient practice. However, it is usually a good approach to

implement a prototype. The second practice is to implement several cards or modules that are connected together with connectors. One example of a product like this is Tower System (Figure 2.2) from Freescale Semiconductor, Inc. Benefit of this approach is that it allows customization of the PMCP by removing or adding modules. It also enables the possibility of designing new modules after the original PMCP design is completed. Disadvantage for this method is increased complexity of the design due to the replaceable modules and higher production costs.

Figure 2.2: Freescale Tower System (Freescale Semiconductor, Inc, 2012b)

Figure 2.3 illustrates the hardware platform for the PCCP. The heart of the hardware platform is a programmable processor that is usually a Reduced Instruction Set Computer (RISC) processor. Usage of the RISC processors, the RISC processor and a Digital Signal Processor (DSP), or the RISC processor and a microcontroller in parallel is also gaining popularity. Usage of multiple processors increases the performance of the system allowing dedication of the functions to a specific processor. This approach is usually less expensive than one complex processor. ARM processor is a popular solution for a processor, but also MIPS-, PowerPC-, and x86-architectures are used (Free Electrons, 2010). The PCCP contains processor i.MX-357 from Freescale Semiconductor, Inc.

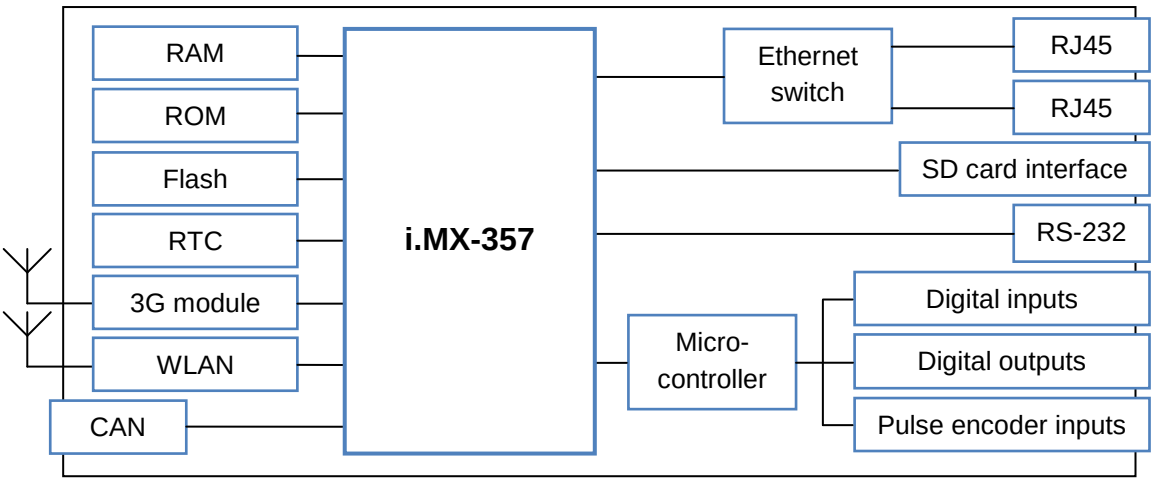

Figure 2.3: PCCP's hardware platform (adapted from (Jocklin, 2011))

Storage elements are included in the hardware platform. These elements can be on-chip (implemented in the processor) or off-chip (outside the processor, at the circuit board or as an external device). Optional memory types are Random Access Memory (RAM), Read Only Memory (ROM), or Flash memory. Obviously, all of these memories can be included in the same platform. In some cases, memory card interface is useful. PMCP can also include chips or microcontrollers for distinct purposes. One example for this kind of chip is a real-time clock (RTC).

PMCP communicates with the external devices such as other machines, PMCPs, and sensors via peripherals. Protocols and buses used for the peripheral communication are Serial (RS-232), digital/analog inputs and outputs, Universal Serial Bus (USB), field bus, Ethernet (RJ45 connector), Wireless Local Area Network (WLAN), 3G, I2C, Serial Peripheral Interface (SPI), Secure Digital (SD) cards, and JTAG. Complex and time-critical functions can be implemented with dedicated hardware accelerators offering faster and more predictable behavior and smaller power consumption (TUT, 2011).

If the PMCP is assigned for specific functions, components and hardware described above are usually selected so that the performance of the PMCP is sufficient to execute the required functions. However, this method sets constraints for the software design and forces the developer to optimize the software performance in more detail. Environment, industrial or home, can also set requirements for the components used at the hardware platform.

2.3 Software platform

One of the most challenging issues with the PMCP is the communication between the hardware and the applications. With the PMCP, this issue is solved by using a software platform layer. This layer contains all software components that are not dependent on the user applications. Instead, the software platform is dependent on the hardware platform. Software platform layer is also known as Hardware-dependent Software (HdS). HdS is the part of the software that directly interacts with the underlying hardware platform (Dömer, et al., 2009). As Figure 2.4 describes, software platform layer (in Figure described as HdS) includes Operating System (OS), communication protocols, device drivers, boot loader, and Hardware Abstraction Layer (HAL). In many cases, OS for the PMCP is Real-Time Operating System (RTOS). One possible RTOS for the PMCP is Embedded Linux.

The purpose of the software platform is to abstract the underlying hardware platform from the upper layers (applications) and realize functions like multitasking and high-level timing. HAL is the software layer that provides an abstract and generic interface to access the hardware resources (Dömer, et al., 2009). For example, OS kernel abstracts processor(s) and memories, drivers abstracts hardware blocks, IO system abstracts peri-

pheral devices, and communication protocols abstracts communication modules. These abstractions allow better portability of the applications and source code for the different PMCPs by modularizing functionalities of complex designs. (TUT, 2011)

In practice, the software platform offers interfaces to access the hardware resources. These interfaces allow the software developer to design applications without knowing unnecessary details about the hardware platform. The developer only needs to know what resources the hardware platform offers and how to use them. Change in the hardware platform requires modifications only to this layer, not into the applications. (TUT, 2011)

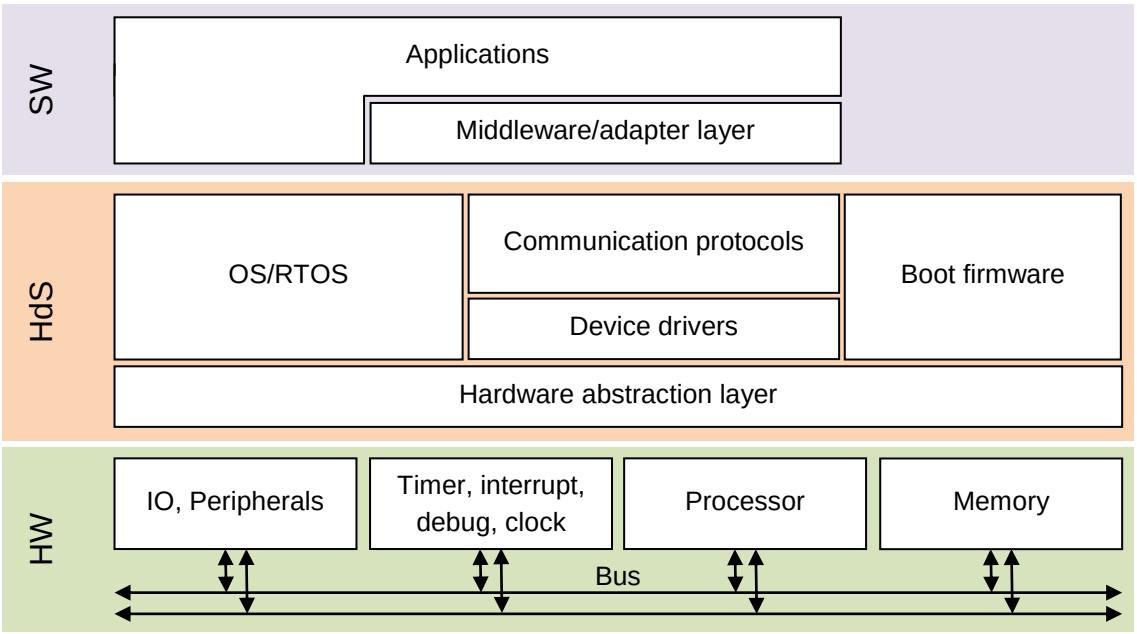

Figure 2.4: Layered architecture (adapted from (Dömer, et al., 2009))

In the best case, the software platform is a powerful and an easy way to access resources of the hardware platform. However, to achieve these benefits, the developer must pay attention to the design and the implementation of the software platform. Software platform offers several benefits such as flexibility, and possibility for a late change and quick adaptability. These benefits are the reason why it has become an important method for designing embedded systems (Dömer, et al., 2009). Furthermore, software layer enables simultaneous software and hardware design and implementation.

Although the usage of the software platform provides huge benefits, it is important to remember the challenges regarding those. Performance of the software layer becomes less efficient when it grows and becomes more general. Verification also becomes more complicated when designing the hardware and software in parallel. In addition, licenses and issues around those should be taken into account when implementing the software platform.

2.4 Embedded Linux

Embedded Linux is a combination of the Linux kernel and open source components developed by the communities of developers and users. These days, Embedded Linux is running in many devices such like televisions, mobile phones, and industrial machines. (Free Electrons, 2012)

Embedded Linux and open source components offers many advantages compared to commercial substitutes. One of the most important features is the ability to re-use components without any costs. Open source community provides components for many standard features like network protocols and graphic libraries. Open source software allows reducing software costs drastically because advanced and free development tools are available. Open source also provides full control for the software since all source code is available to the developer. This makes it easier to make modifications, do debugging and optimize the system under development. Quality of the open source components is usually high because testing of the most-widely used components is quite comprehensive. Availability of the open source components also allows easy exploration of new solutions and helps avoiding time-consuming purchasing and demonstration procedures. (Free Electrons, 2012)

In Figure 2.4, Embedded Linux realizes OS/RTOS, communication protocols, and device drivers at the layer of the HdS. Benefits listed above make Embedded Linux a well suitable operating system for the PMCP. Although several other OS options exist, this Thesis concentrates only on Embedded Linux. Multiple different Embedded Linux variants are available. Ready to use options are for example uCLinux and Android. Embedded Linux is also possible to be configured and built manually by the developer. This option is related to the embedded software frameworks, which are introduced later.

2.5 Applications

In the case of the PMCP, applications are used by the user of the system or by other applications. Applications should not be dependent on the hardware platform. Applications should not directly access the hardware platform. Instead, applications should use the interface offered by the software platform. If application accesses the hardware platform directly, problems may occur if the hardware platform changes at some point of the PMCP's life cycle.

As the name PMCP implies, applications can be developed with several programming languages. These languages can be C, C++, Python, or something else. Available programming languages are case dependent and chosen by the system developers.

In the case of the PMCP, constraints might affect the applications. Performance, memory, and real-time issues must be considered when designing and implementing an application. It is important to understand that running multiple heavy-duty applications simultaneously might cause delays in response times. Requirements for the response times play an important role with the PMCP. Controlling of an industrial machine requiring high performance, safety, or other properties requires strict boundaries for the response times. Measurements to improve software quality and estimate maintenance needs and workload with the PCCP was investigated by Jocklin (2011).

3 SYSTEM DEVELOPMENT AND LIFE CYCLE MANAGEMENT

3.1 Introduction

It is important to introduce basic methods and processes used in the system and software development to understand specific problems concerning the life cycle management. After this, introduction to methods for the life cycle management follows.

Definition of the software development

> *"The process by which user needs are translated into a software product"* (IEEE 24765-2010, 2010).

Definition of the life cycle

> *"The evolution of a system, product, service, project, or other human-made entity from conception through retirement."* (IEEE 828-2012, 2012).

The life cycle of the system or the software development project includes everything between the beginning and the ending of the project. In practice, this means the life cycle of the system begins at the idea that something should be done. Vice versa, the life cycle ends when all activities supporting the system are finished and all data related to the system is archived. A system life cycle model (Figure 3.1) introduces six phases carried out during the system's life cycle (IEEE 24748-1-2011, 2011). These phases are concept, development, production, utilization, support, and retirement. The system life cycle model includes both hardware and software designing.

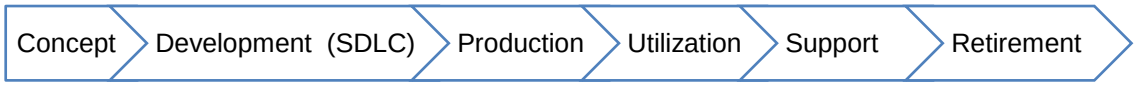

Figure 3.1: System life cycle model (IEEE 24748-1-2011, 2011)

During the concept phase, user needs for the system are determined and the system concepts are described. The development phase is specified as a process called Systems Development Life Cycle (SDLC) described in Figure 3.2. SDLC is described later in this Chapter. During the production phase, the designed system is assembled, tested, and deployed. The utilization phase contains operations to maintain the system and make improvements to it. During the support phase, customers are given support on operating the system effectively. Finally, during the retirement phase system is stored, archived or disposed. (IEEE 24765-2010, 2010) (IEEE 24748-1-2011, 2011)

The SDLC process includes five phases: analysis, design, implementation, testing, and evaluation (Figure 3.2). The analysis is a process to partition the specified system into functions, components, or objects and define relations between those. During the design phase, architecture, components, and all elements of the system are defined, documented, and verified against requirements. Decisions made during the design phase are realized at the implementation phase. During the implementation phase, the system is created. The testing phase contains operations to evaluate that the system fulfills the requirements defined earlier. The purpose of the evaluation phase is to evaluate that the system satisfies all necessary standards, regulations and guidelines. (IEEE 24765-2010, 2010)

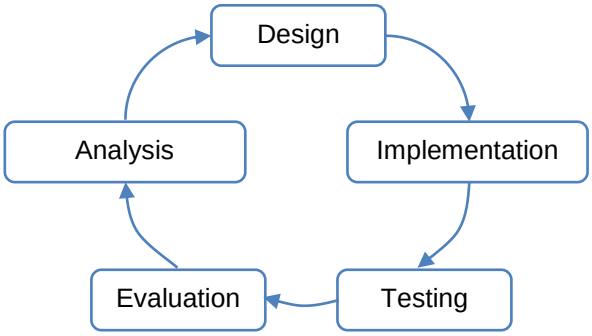

Figure 3.2: Systems development life cycle (SDLC)

The system life cycle model and the SDLC are also valid for the PMCP. The life cycle management is a process to control all of the six phases throughout the system's life cycle. It is important to notice that the execution of the phases described in the model may be overlapping.

A great amount of data is gathered throughout the project's life cycle. The data consist of metadata, documents, source codes, test plans, and executables. All this data must be stored to a secure location in order to ensure accessibility at any time through the project's life cycle.

Managing different products and software that are evolving is not an easy task. Usage of processes maintaining these issues plays an important role. Product Life cycle Management (PLM) and Application Life cycle Management (ALM) are introduced in Sections 3.3 and 3.4. These are used to assist with controlling data, tasks, personnel, and project management. Introduction of the Software Configuration Management (SCM) is given in Chapter 4.

The relation between the development models, PLM, ALM, and SCM is illustrated in Figure 3.3. As the figure presents, all of these methods are related to each other. Methods even share objectives and tasks but handle them from a different perspective. Haikala et al. (2006) argue that software product development is a cyclic process. New software features are added, existing bugs fixed, and customer feedback taken into ac-

count. Life cycle management processes, ALM, PLM, and SCM are used to control these tasks.

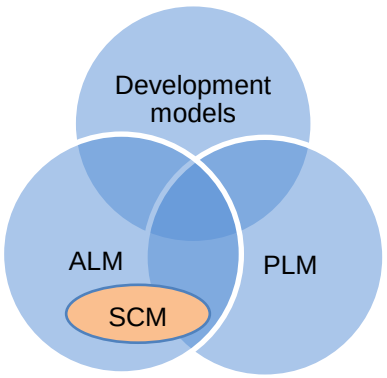

Figure 3.3: The relation between PLM, ALM, SCM and development models

Developing software for the PMCP differentiates slightly from the normal software development. As mentioned previously, industrial machines set requirements for the software. Clear guidelines and maintainable life cycle management process play an important role in controlling those requirements and maintaining the PMCP throughout its life cycle.

3.2 Software development models and activities

3.2.1 Activities

The system life cycle model describes phases within the context of the system development. This Section introduces activities (also known as phases) related to the software development. Software development activities are categorized in five sections: requirements, design, implementation, verification, and maintenance.

The purpose of the requirements phase is to determine and document all requirements for the software product. The design phase includes operations regarding the definition of architecture, software, interfaces, and all essential parts concerning the software product. Definitions are documented and verified against requirements. In the implementation phase, the software is created based on the design documentation. During the verification phase, software evaluation is carried out to determine that all requirements specified in the beginning of the project are fulfilled. (IEEE 24765-2010, 2010)

Maintenance can be corrective, adaptive, or perfective and these operations are performed after the delivery. Corrective maintenance is modification of the software to correct defects, faults, or problems. Adaptive maintenance is modification of the software to maintain its usability in a changing environment. Perfective maintenance is an action performed for the software to improve performance, maintainability, and functionality or correcting faults that are not yet classified as failures. (Leon, 2005) (IEEE 24765-2010, 2010)

3.2.2 Waterfall model

The Waterfall model is a commonly used model for a software development process. The model contains phases (activities) performed in a specific order without any iteration. Some overlapping may occur with these phases. Figure 3.4 illustrates the waterfall model, which consists of five distinct phases: requirements, design, implementation, verification, and maintenance. (IEEE 24765-2010, 2010)

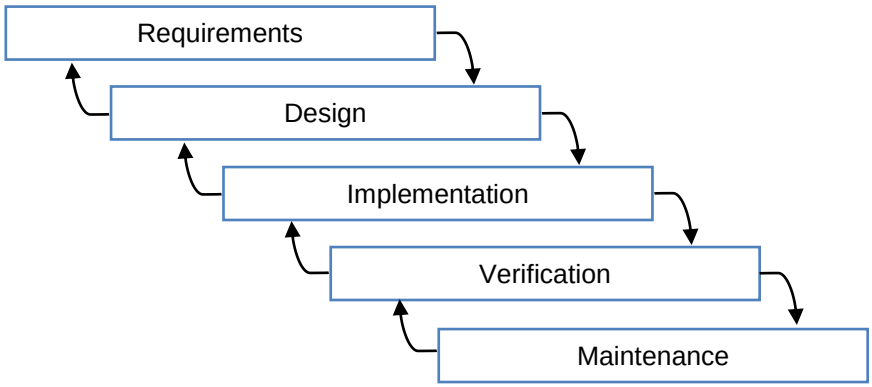

Figure 3.4: Waterfall model

3.2.3 The V model

The V model resembles the waterfall model. Main difference is that after one phase is executed, it must be verified before starting the next phase. The result of the previous phase is used as a base for the next phase. Figure 3.5 illustrates the V model.

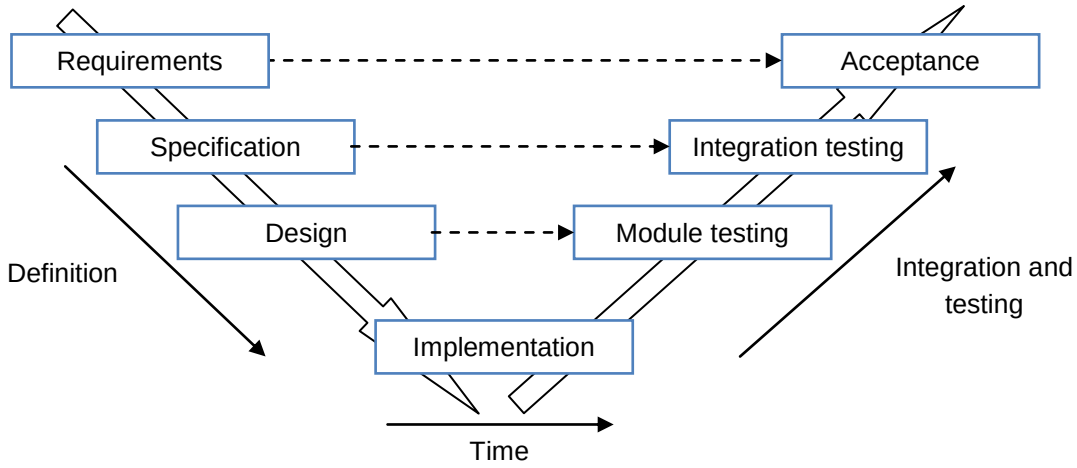

Figure 3.5: The V model

3.2.4 Prototyping

A prototype is used to test new features and technical solutions, get user feedback, or to find out possible customer requirements. Prototyping model is used for both hardware and software. There are two options how functioning prototype is utilized. First option

is that the requirements for the actual system, which is implemented from scratch, are determined based on the prototype. This approach is illustrated in Figure 3.6. Second option consists of using prototype as a base which is then developed as a complete system. (Haikala, et al., 2006) (IEEE 24765-2010, 2010)

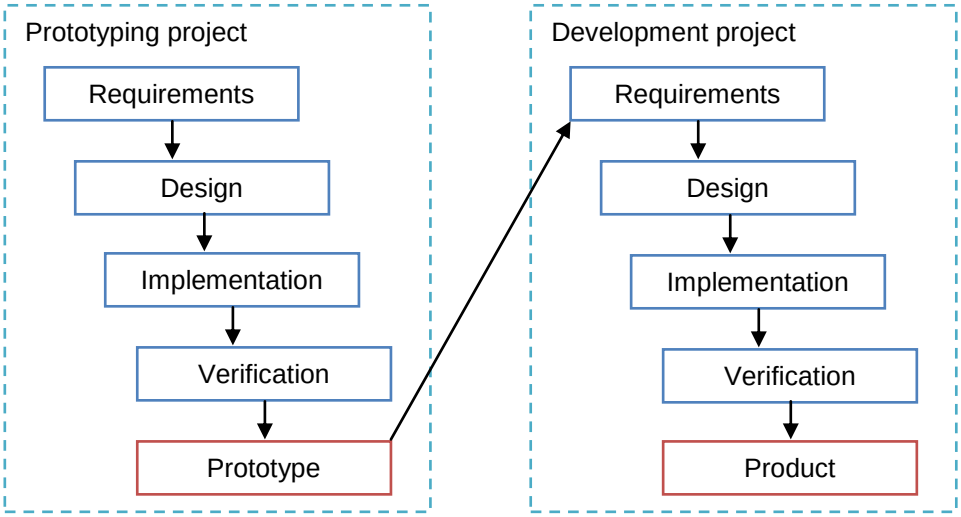

Figure 3.6: Prototyping model (adapted from (Haikala, et al., 2006))

Usage of prototypes offers the possibility to make performance and user interface testing (Haikala, et al., 2006). In the case of machine control, prototyping is a useful method to test capabilities of the device or software in a real, controlled environment. Gathering information from the usage of the prototype allows more accurate decision-making. Prototyping development model is used with the PCCP.

3.3 Product life cycle management

Definition of Product Life Cycle Management (PLM)

> *"PLM is the activity of managing company's products all the way across their lifecycles in the most effective way"* (Stark, 2011)

Kääriäinen et al. (2009) argue that the PLM is a system with multiple subsystems for creating and managing product related data. Subsystems might be needed to be integrated in order to ensure the PLM system to work correctly. Such subsystems for the PLM include Software Configuration Management (SCM), resource management, and Enterprise Resource Planning (ERP). In this Thesis, the PMCP is interpreted as a larger entity, which includes many products that are controlled and managed throughout their life cycle. The PLM is responsible for controlling the system life cycle process.

3.4 Application life cycle management

Definition of Application Life cycle Management (ALM)

> *"The coordination of activities and the management of artifacts (e.g. requirements, source, test cases) during the software product's life cycle"* (Kääriäinen, et al., 2009)

Depending on the source, definition of the ALM is more or less tool-focused. As the PLM is responsible for managing the products, the ALM is responsible for managing the software portion of the product. In practice, this means that ALM is controlling the SDLC process. However, because sometimes the software portion is the entire product, PLM and ALM can be overlapping. Configuration management has significantly influenced ALM during its conception (Kääriäinen, et al., 2009). For this reason, SCM, which is related to the ALM, is introduced in Chapter 4.

Lacheiner et al. (2011) define the objective for the ALM as "to provide a comprehensive technical solution for monitoring, controlling, and managing software development over the whole application life cycle". Traditionally tool vendors have promoted ALM with their own perspectives to market their own solutions for the ALM. This has led to the situation where ALM has slightly different meaning in different situations or by different vendors. (Lacheiner, et al., 2011)

Chappell (Chappell, 2008) divides ALM into three areas: governance, development and operations. Governance contains project management and decision making through the entire life cycle of the application. Development includes the creation of the application and it can be performed multiple times during the life cycle. Operations are the tasks needed to manage and run the application until the end of its life cycle. These three areas overlap and have an effect on each other. (Chappell, 2008)

3.5 Summary

Figure 3.7 illustrates how company's products are related to the PCCP, PLM, and ALM. Applications running on the PCCP are varying between different products. PLM is the life cycle management process for the products. Correspondingly, ALM is the life cycle management process for the applications used in the products. In this Thesis, designed processes are mainly used to control the ALM. Although as presented in Figure 3.7, ALM can be considered to belong to the PLM.

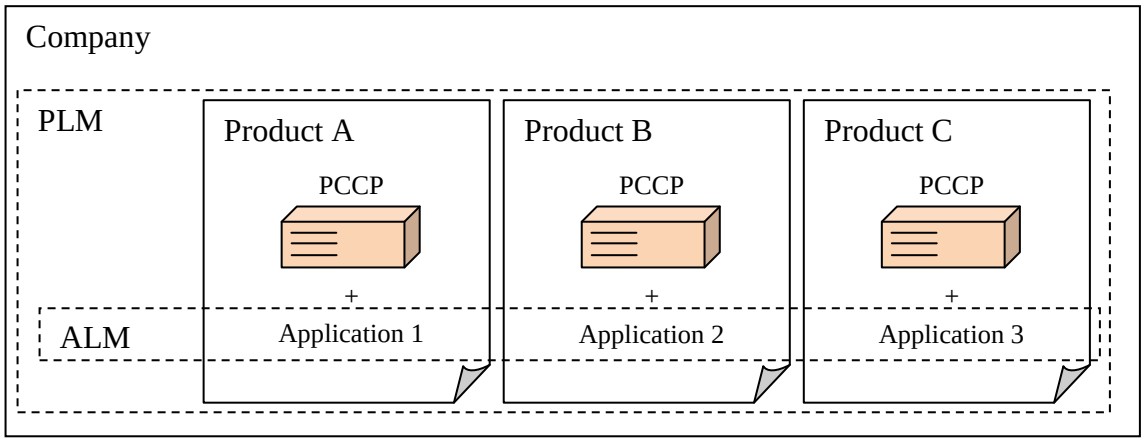

Figure 3.7: PCCP, PLM and ALM

4 SOFTWARE CONFIGURATION MANAGEMENT

4.1 Introduction

"Change is inevitable in all projects, and unmanaged and uncontrolled change is trouble all the way" (Leon, 2005)

PMCP includes hardware and software platforms and a great number of heterogeneous applications, which are constantly changing and dependent on other resources. Without proper methods for controlling these items during the system's life cycle, irreversible damage might happen.

Definition of Software Configuration management (SCM)

> *"the discipline whose objective is to identify the configuration of software at discrete points in time and the systematic control of changes to the identified configuration for the purpose of maintaining software integrity, traceability, and accountability throughout the software life cycle."* (Leon, 2005)

Software Configuration Management (SCM) is a process for controlling configurations of the software product. Purpose of the SCM process is to identify items under configuration management, establish and maintain integrity of the items, and make the items available to essential personnel and organizations throughout the life cycle of the software (IEEE 12207-2008, 2008). Although the name of this method only concerns software, it is important to notice that information related to the hardware platform can also be brought to the SCM. Furthermore, the SCM allows all PMCP related information to be gathered to one system.

The valuable SCM system is required to offer reproducibility, stability, auditability, traceability, scalability, control, and safety. In this Thesis, these attributes are called requirements for the SCM system. Reproducibility determines that every task, process, or build under the SCM must be able to be reproduced in the same way as at the first time it was produced. Continuous and correct functioning of the SCM system is called stability. Auditability determines that all necessary information regarding software, builds, and other objects must be collected to aid decision-making. Traceability is the ability to track changes made during the software development. Scalability describes the SCM system's ability to support software projects of different sizes and methods. Control determines that the SCM system is managing all items under the configuration

management system and ensures that changes are applied to the items. Safety determines that in a case of failure, the SCM system is able to recover. Safety also defines that only personnel with authorization are permitted to operate the system. (Elemo, 2008)

Benefits of the SCM (Keyes, 2004) (Leon, 2005)

- Better integrity of the software and process
- Better control on the software development activities
- Higher productivity of the software development
- Lower amount of bugs and defects
- Improved quality
- Increased reuse of the software
- Ability to track changes
- Data stored to a version control system allowing software versioning
- Ability to trace requirements
- Reduced costs of the software maintenance
- Ability to reproduce previous implementations
- Improved utilization of the organizational resources

Because of personnel changes, software and system development should not be dependent on the personnel. Instead, development work should be dependent on the process (Leon, 2005). This means that tasks and processes should be documented properly to allow any developer to execute them.

4.2 The SCM system and process

Schwaber (2005) argues that ALM solutions are based on the SCM tools, which provide storage, version control, and traceability for the objects under the SCM throughout the life cycle of the product. Typically, SCM includes tasks for process and build management, software versioning, release engineering, defect tracking, and configuration item controlling. Figure 4.1 illustrates processes related to the SCM.

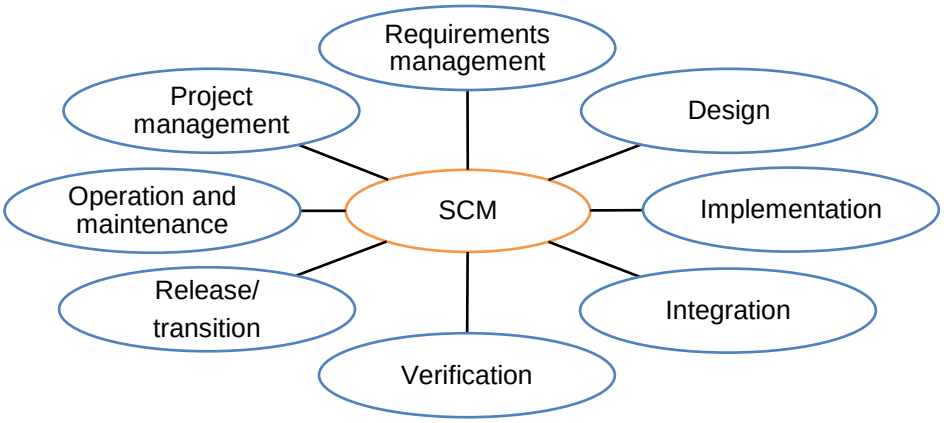

Figure 4.1: Processes for the SCM (adapted from (IEEE 828-2012, 2012))

Haikala et al. (2006) divides the SCM into three sections: components, configurations, and procedures. Figure 4.2 illustrates these sections with their main tasks. Although this partition is rather simplified, it presents the SCM in a way that is easy to understand. (Haikala, et al., 2006)

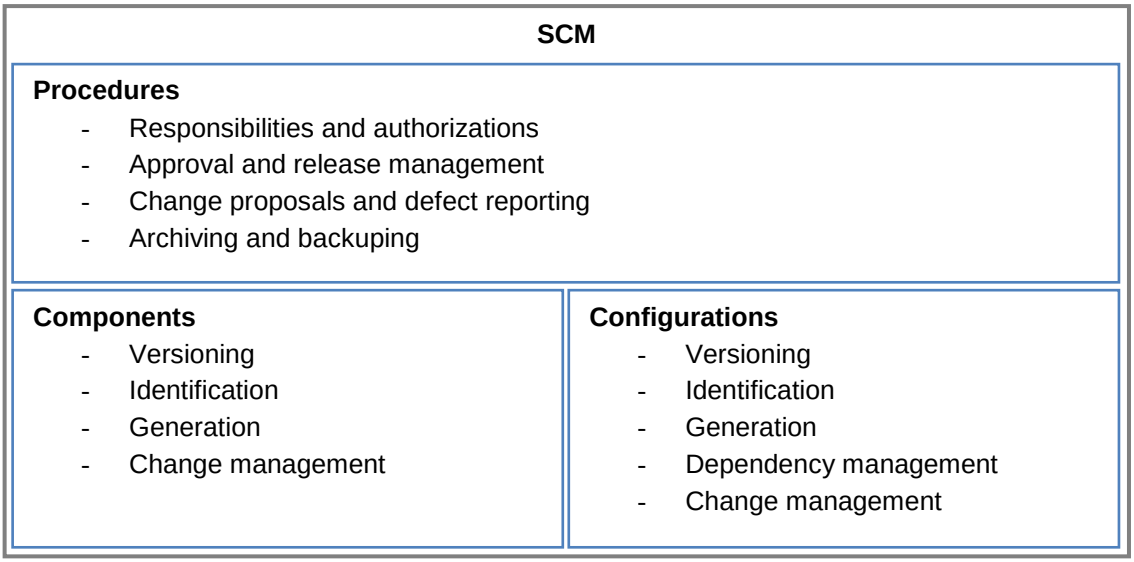

Figure 4.2: SCM (adapted from (Haikala, et al., 2006))

IEEE standard defines outcomes for a successful implementation of the configuration management process. These outcomes are that configuration management strategy and items requiring configuration management are defined, configuration baselines are established, and the status of items is made available throughout the life cycle of the software. In addition, changes to the items under the configuration management and the configuration of released items are controlled. (IEEE 12207-2008, 2008)

SCM implementation varies between different companies and projects so there is no simple guideline for the implementation process. Leon (2005) argues that the software development process and procedures should be integrated with the SCM to achieve maximum benefit for the SCM implementation. IEEE standards are a useful source when implementing a SCM system.

4.3 Concepts

To understand SCM phases and activities, it is first necessary to know basic concepts related to the SCM. In Table 1, concepts used with the SCM are described briefly.

Table 1: SCM concepts (IEEE 24765-2010, 2010), (IEEE 828-2012, 2012), (Leon, 2005)

Concept	Description
Baseline	Approved, reviewed, and fixed version of the configuration item or set of configuration items. Agreed basis for coming development activities. Changing is not allowed without special procedure.
Branch	Deviation from the planned development line. This occurs when doing bug fixes, improvements, or modifications to previous versions of software.
Change Control Board (CCB)	Also called a Configuration control board. Group of people whose function is to approve or disapprove change proposals for the configuration items.
Check-in	Putting configuration item back to the repository after modifications are done.
Checkout	Taking configuration item from the repository to make modifications.
Configuration item (CI)	Identifiable hardware or software component that is constantly changing and thus taken under the configuration management. Typical CIs are source code modules, metadata, specifications, programs, libraries, documents, test cases, test plans, compilers, tools, and development procedures.
Delta	The difference between the new and the previous version of the module.
Milestone	Scheduled point or event during the project. Used to track progress of the project.
Release	Full or a part of the software product or a set of configuration items that are distributed for the customers. Release is tested and certificated to fulfill all defined requirements, specifications, and constraints.
SCM database	Repository for storing information regarding the configuration items and relations between the configurations items.
System build	Standalone and functional version of a system, which is generated for the target platform. System build process can be automated with the SCM tools.
Tag	Descriptive name given for a release or a branch.
Trunk	The main development line of the software. Branches distinguish from the trunk.
Variant	Version of a software generated to increase fault tolerance.
Versioning	Assignment of a unique version name or number for the configuration item.

4.4 Phases

Leon (2005) introduces ten phases for the SCM process. Figure 4.3 illustrates the SCM phases described as a workflow in a chronological order with the related system life cycle phases. Phases with descriptions are listed below in the order of execution (Leon, 2005).

1) SCM system design
   SCM system design team is selected. Team defines the purpose of the SCM system, activities, tools, versioning, release management, and many other things. Team determines how these tasks are performed and what tasks will be automated.
2) SCM plan preparation
   System details determined during the SCM system design phase are recorded to a document called SCM plan. The SCM plan is distributed to all personnel using the SCM system.
3) SCM team organization
   SCM team is selected and responsibilities are given to personnel. Change control board is selected.
4) SCM infrastructure setup
   Infrastructure to assist the SCM system and team is formed.
5) SCM team training
   SCM team is trained to operate with the SCM system and specific project guidelines for the SCM are given.
6) Project team training
   Project team is trained to operate with the SCM system. Training is a continuous task running throughout the product's life cycle.
7) SCM system implementation
   Installation of the SCM tools and assignment of the responsibilities for the personnel are accomplished.
8) SCM system operation and maintenance
   SCM activities are executed and the SCM system is maintained.
9) Records retention
   Records and documents in the SCM system are archived, retained, or destroyed. This is done before retiring the software.
10) SCM system retirement
    The SCM system is retired and the personnel are released from the project.

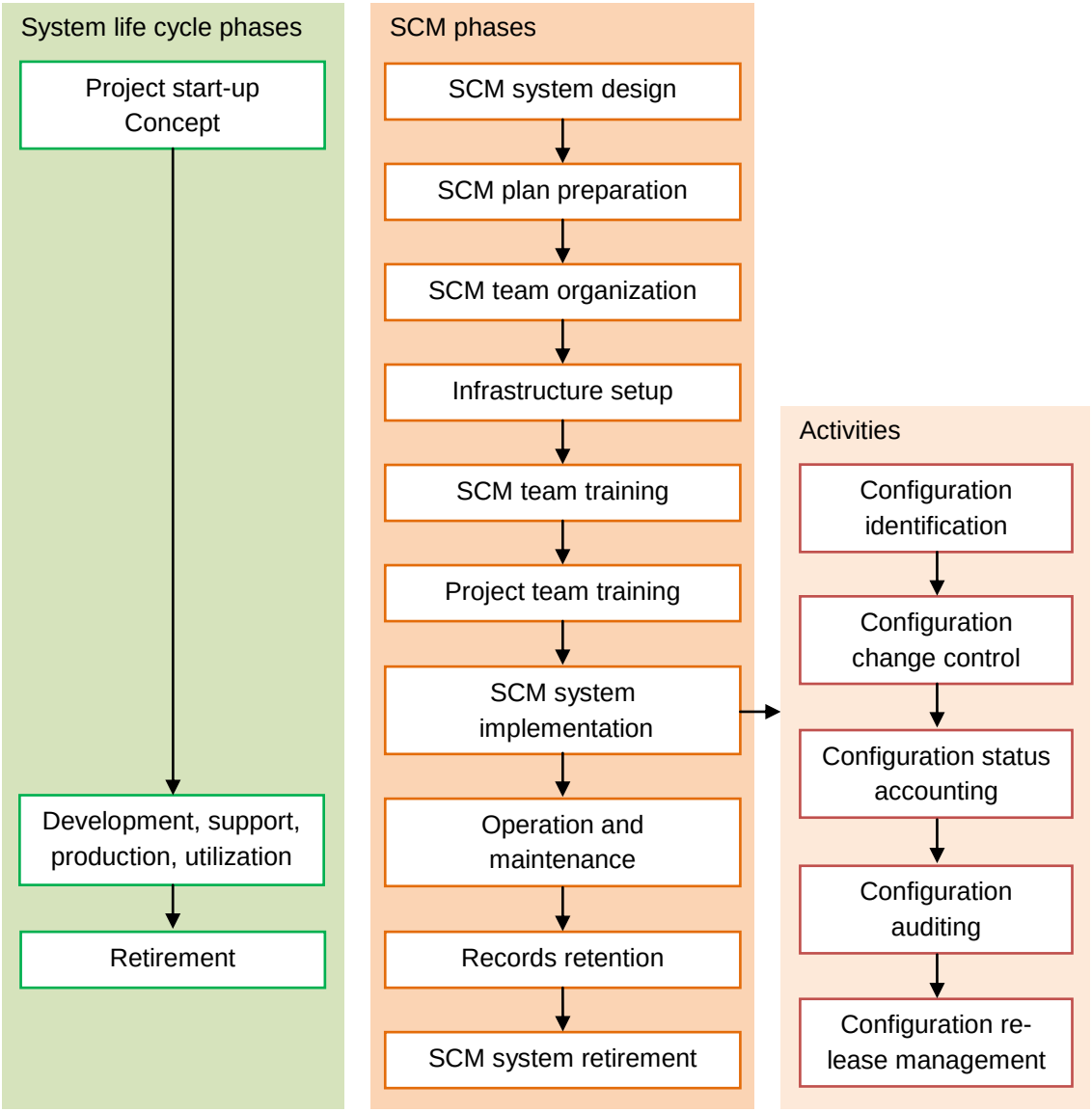

Figure 4.3: SCM implementation phases (adapted from (Leon, 2005))

4.5 Activities

IEEE standard (IEEE 828-2012, 2012) introduces five specific activities for the SCM. Activities are configuration identification, configuration change control, configuration status accounting, configuration auditing, and configuration release management (Figure 4.4). The purpose of the activities and outcomes for each activity based on the IEEE Standard 828-2012 is described in Table 2.

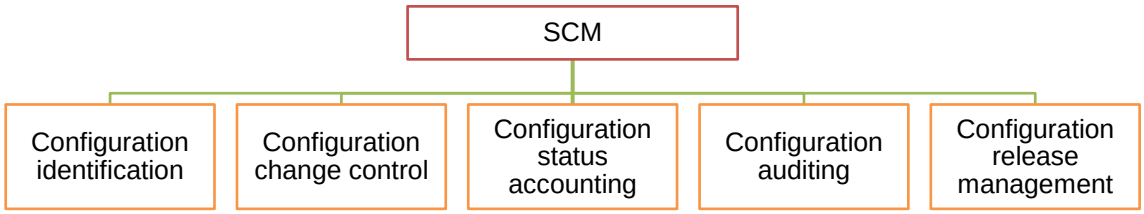

Figure 4.4: SCM activities

Table 2: Configuration management activities (IEEE 828-2012, 2012)

Configuration identification	Purpose	Identify and set names for the configuration items.
	Outcomes	Configuration is defined, configuration items are identified, and baselines are generated.
Configuration change control	Purpose	Control changes during the life cycle of the product.
	Outcomes	Requests for changes to the configuration items are processed. Approved changes are realized and verified. All changes are archived.
Configuration status accounting	Purpose	Share and store information about the configuration items and all information concerning those.
	Outcomes	Needed information and reports are identified and accessible for all relevant personnel.
Configuration auditing	Purpose	Evaluate the product and its changes to make sure that the requirements are fulfilled and that functional properties are valid.
	Outcomes	Audit practices are defined, audition for the product is carried out, all information concerning the audit is stored and actions based on the audit findings are identified.
Configuration release management	Purpose	Deliverables are delivered to the customer (or some other) in agreed format at specified time.
	Outcomes	The content and the format of the release are defined, deliverables are delivered and information about the releases is shared.

4.6 Documentation and tools

Documentation of the SCM should be started at the beginning of the project. The most important document to be created is the SCM plan. This document describes all necessary information related to the SCM: system design, procedures, tasks, and responsibilities. By reading this document, the developer should be aware of all key information required to know about the implemented SCM system. (Leon, 2005)

Other typical documents related to the SCM are project plans, test plans, reports (problem, change), forms (inspection, review), and requests (service, change) (Keyes, 2004). Used document types are project specific. The purpose of the documentation is to keep all vital information in an agreed format. Usage of reports, forms, and requests also improves transparency and helps personnel to share information easily.

SCM tools have existed for a long time. The importance of the SCM tools is growing as the complexity of the systems is increasing. SCM tools are used to control and automate

the SCM process. It is not an easy task to select an appropriate SCM tool from variety of possible candidates for the specific project.

First SCM tools were used only to control changes in the source code (Leon, 2005). Modern SCM tools can handle various tasks during the SCM process. SCM tools include features like requirement tracking and management, defect tracking, build management, release management, software controlling, test management, and quality assurance. SCM tools also help to automate these tasks and improve development quality (Leon, 2005). Several open source and commercial options for the SCM tools are available.

Integration of the SCM tools with the existing tools and systems is a major issue when choosing the SCM tool for the project or company. Capabilities of the SCM system cannot be utilized without proper connections to version control systems, integrated development environments, test automation frameworks, and other tools.

4.7 Summary

During this Thesis, SCM implementation is not realized as extensively as described in this Chapter. Major tasks related to the SCM are planned and implemented. These tasks are versioning of the components and configurations, dependency and change management, and archiving and backupping. Realized tasks are highly related to the embedded software frameworks. This Thesis discusses only how technical aspects of the SCM system can be implemented. Other aspects, like forming the SCM organization, are not included in this Thesis.

5 EMBEDDED SOFTWARE FRAMEWORK

5.1 Introduction

Building a target system, an embedded Linux distribution, or embedded software for the PMCP is a complex process. This process can be done manually, using distributions or ready file systems, or using embedded software frameworks (Free Electrons, 2012). Although it is a beneficial skill for the developer to know how to do this process manually, it is not recommended with large and complex system implementations. Better practice is to use embedded software framework, which is a toolset or software providing the capability to automate this process. Embedded software framework is not a commonly used name for these tools. This name is used in this Thesis to increase readability and to offer a describing name for these tools.

Definition of a framework

> *"A reusable design (models and/or code) that can be refined (specialized) and extended to provide some portion of the overall functionality of many applications or a partially completed software subsystem that can be extended by appropriately instantiating some specific plug-ins."* (IEEE 24765-2010, 2010)

In the context of definition above, embedded software framework is a reusable and configurable model that contains a large number of packages and it can be extended by implementing specific software or adding new features. Embedded software framework provides mechanisms and methods to make the build process more controllable. Furthermore, it provides means to fulfill some basic requirements for the SCM.

A properly configured and used embedded software framework offers: tools and processes to maintain control and stability, allows scalability of the PMCP under design and enables reproducibility and traceability. From the developer's point of view, an embedded software framework is a valuable tool allowing the automation of the build process. It also simplifies the work of controlling and managing the software and hardware configurations. Implementation and configuration of the embedded software framework is not an easy task and it requires careful planning and testing.

Selecting an embedded software framework for a specific project is a case-sensitive decision. Different projects might use different hardware and software platforms, which sets restrictions for the embedded software framework. In Section 5.7, comparison of the frameworks is carried out within the context of the PCCP. All features, tools,

processes, and concepts related to the embedded software frameworks are introduced in this Chapter.

Although frameworks are licensed under specific licenses, it is important to notice that packages that are built with the embedded software framework may be licensed differently. The developer should always check these licenses to ensure that the system built and distributed would not cause any license issues. (Free Electrons, 2011)

5.2 General features of frameworks

Embedded development environment (Figure 5.1) is a system including embedded software framework and other essential tools. Developer uses this environment to implement the designed system and to build, test, and modify it. This system includes a PC connected to the Internet, VCS, and PMCP.

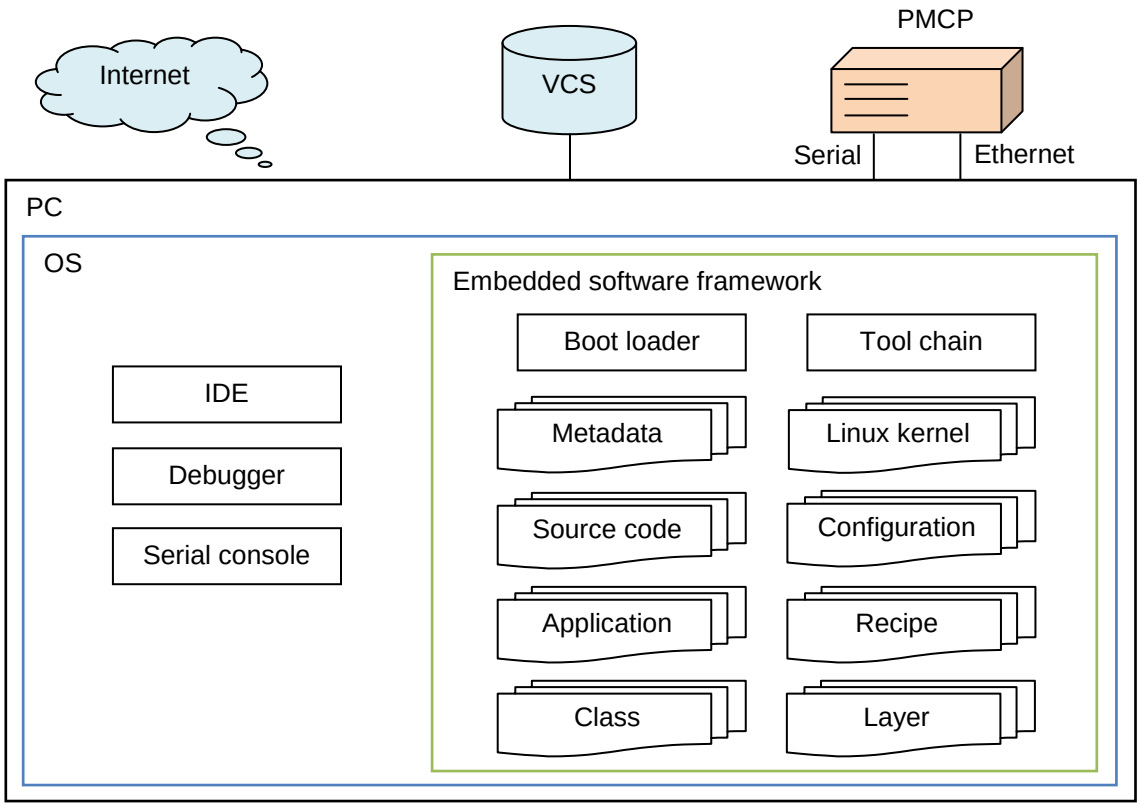

Figure 5.1: Embedded development environment

The developer chooses the operating system (OS) for the embedded development environment. However, embedded software framework can set some requirements for this selection. This is because all frameworks will not work or are not tested thoroughly on all operating systems. Otherwise, OS can be Windows, Linux, Linux virtual machine on Windows or something else. However, if the target PMCP contains embedded Linux, then it would be preferable that the OS for the embedded development environment would also be Linux. This is because then the developer uses the same OS that is run-

ning on the system under development. This can help to avoid a lot of problems during the project's life cycle. One common issue is the compatibility with character encoding.

Embedded development environment includes all tools needed in the system development process. Environment usually includes tools like Integrated Development Environment (IDE), embedded software framework, tool chains, and debuggers. Development work for the PMCP is often called a cross-platform development (Free Electrons, 2012). This means that the system architecture on the development computer differentiates from the actual target architecture.

User interface for the embedded software framework is textual, graphical or a combination of both. Open source frameworks usually have a command line interface; however, plugins for the IDEs allow usage of the frameworks from a graphical tool. In addition, some open source frameworks, like the Yocto Project, have introduced simple graphical tools for the configuration of the framework. Commercial frameworks usually offer both command line and graphical tools. Common concepts related to the embedded software frameworks are introduced in Table 3. Naming of the particular concepts may differ between the embedded software frameworks. These concepts are related to the Yocto Project, but are also valid for other frameworks.

Table 3: Embedded software framework concepts (Yocto Project, 2012b)

Concept	Description
Append file	File that is used to append or modify a recipe from other layer.
Board Support Package (BSP)	BSP is a set of instructions, configuration files, and recipes. These describe features and capabilities of a specific hardware.
Build directory	Directory that contains all files used and generated by the build system. Developer can create several build directories and it is recommended to use different build directories for different target architectures.
Class	Common patterns encapsulated into one file allowing recipes to inherit those when needed.
Configuration files	Determines build options and defines configurations for the hardware platforms.
Distro	Configuration file for the distribution. Contains information about used tool chain and determines globally used variables.
Image	Single binary file, which is compiled based on the recipes. Images will run on the target architecture.
Image recipe	Recipe file determining contents of an image.
Layer	Layers categorize and divide system to the abstraction levels. Layers can contain machine configurations, Linux kernel recipes, boot loader recipes, image recipes and application recipes.

Metadata	Includes recipes, configuration files, and classes.
Package	Compiled application or image binary.
Recipe	A simple text document that contains a set of instructions for building binary packages and images. Recipe includes all necessary information: configuration and compilation settings, location of the source code, needed patches, and dependencies for the libraries and other recipes. Recipe can be used to determine configurations and build settings for many purposes: applications, Linux kernel, boot loader, and root file system images. In this Thesis, recipes are categorized as application, image, image baseline, or image release recipes. Layout for all recipes is similar, only the purpose of the recipe differs.
Root file system	Includes all files needed to run the Embedded Linux on the target PMCP. Root file system is placed in the memory of the PMCP.
Tool chain	Set of development tools and utilities to develop software for the target architecture. Tool chain contains cross-compilers, linkers, and debuggers. Embedded software frameworks are usually delivered with several tool chains each having specific target architectures like ARM, x86, PowerPC, or MIPS. Tool chain generates code for the target device, but runs on the development machine.

Layers (or other similar structure) are usually used to describe the structure of the embedded software frameworks. Figure 5.2 illustrates the layer structure used with the Yocto Project. As the figure presents, the layers are divided into logical segments. This approach enables layers to be modified and created by different individuals or companies. The developer can choose what layers to use, create new layers, or even create an own layer structure if necessary. (Yocto Project, 2012b)

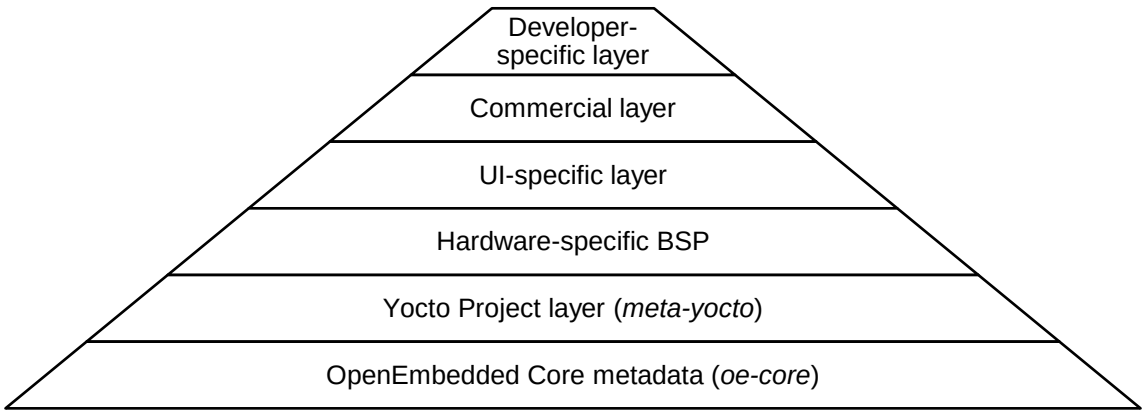

Figure 5.2: Yocto Project layers (Yocto Project, 2012b)

Embedded software framework automatically downloads, configures, compiles, and installs all necessary source packages to build an application, root file system, or other images. Generally, embedded software framework is responsible for building the tool

chain, root file system, and boot loader and Linux kernel images. The developer can specify the Linux kernel version for the Embedded Linux; although not all features of the processors are supported in all kernel versions. The developer can also select the tool chain used to execute the build process. (Free Electrons, 2011)

Figure 5.3 describes a general workflow for the execution of the build process with the embedded software framework. Configuration of the embedded software framework is usually carried out by creating or selecting a BSP and by modification of configuration files and recipes. The build process produces packages for the applications and images for the root file system, boot loader and Linux kernel. Below is listed the order of installation, configuration, and building with the embedded software frameworks (Yocto Project, 2012b).

1) Install and prepare OS
2) Install embedded software framework
3) Initialize build environment
4) Set build configurations/choose BSP
5) Modify or add recipes
6) Choose recipe or image to built
7) Configure Linux kernel (optional)
8) Configure Boot loader (optional)
9) Build
10) Repeat steps 3-9

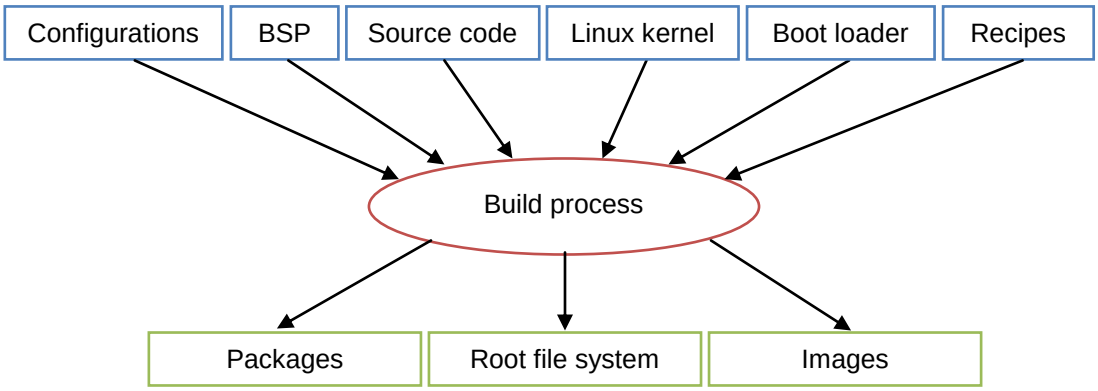

Figure 5.3: General workflow

Typically, embedded software framework includes BSPs for several known hardware platforms. If the developer cannot use these with the PMCP under development, the existing BSP is modified or a new BSP is created.

In these days, importance of the software licensing has become a major issue. The developer must consider issues with the licensing when implementing software for the PMCP. Embedded software frameworks offer tools to monitor and control licenses involved with the system implemented.

5.3 Tools used with a framework

5.3.1 Version control system

A Version Control System (VCS) is probably the most important tool used with the embedded software framework. The VCS is used to fulfill two basic requirements for the SCM: reproducibility and control. Reproducibility is partially fulfilled by adding all essential software components (configuration items) to the VCS. This allows older versions of the components to be retrieved. Furthermore, VCS can be used as a storage and backup system, which enables components to be controlled.

The VCS can be used by the developer or by the embedded software framework. Embedded software frameworks retrieve data, such as Linux kernel sources, packages, and patches from the VCS. This practice allows the usage of the latest component versions. The developer can store applications, recipes, documents, and other metadata to the VCS. Popular choices for the VCS are GIT and Subversion (SVN).

5.3.2 Integrated development environment

Integrated development environment (IDE) is software that helps and automates the developer's software development tasks. The IDE includes tools for source code editing, debugging, compiling, and build automation. Integration of the embedded software framework to the IDE allows developer to control the build process, software configurations, VCS, issue and bug tracking, and the target PMCP from a single user interface.

Eclipse is a popular open source IDE. Eclipse is extended by using plugins provided by companies, communities, or individual developers. Many commercial products are based on Eclipse. (Eclipse Foundation, 2012)

5.3.3 Automated build system

Embedded software framework executes the software build process for the PMCP. Without any additional tools, the build process is executed only by the developer's commands. Automation of this build process provides many advantages. Regular and scheduled builds assist the developer in finding problems in the built software. The automated build system combined with automated test procedures aid in finding bugs after changes are applied in the software. The results of the build and test process are reported for necessary personnel during and after the automated build process by email or by using web pages.

5.3.4 Continuous integration

Automated build system described above is realized by using the embedded software framework with tools for continuous integration. These tools provide functions for ex-

ecuting the build process, running automated tests after the build process, and generating reports and statistics about the build and test processes. Together with other tools, continuous integration provides an automated and controlled way to execute the build process.

5.3.5 Issue and bug tracking

Issue and bug tracking tools play an important role in implementing traceability and control. These tools provide an easy way to document and log: issues, bugs, releases, and tasks. Usage of the issue and bug tracking tools also helps project management.

5.3.6 Verification and analysis

Verification and analysis of the developed software and hardware is an important task. Problems and bugs in the components might not be discovered without proper mechanisms, processes and tools. Usage of the unit and integration testing tools and static code analysis with the embedded software frameworks allows verification and analysis to be performed automatically, simultaneously, and effectively. Software quality measurements for the PCCP were developed by Jocklin (2011).

5.4 Usage of framework for software configuration management

Setting up the embedded software framework is performed by using BSPs, recipes, and configuration files. This method offers an easy starting point for selecting configuration items. BSP realizes the hardware abstraction layer. Adding BSPs to the software configuration management, maintenance of the hardware platform becomes easier.

Structure of the embedded software frameworks helps the maintenance process of the software. Applications are described in individual recipes, which allow maintenance of the application versions. A single recipe is interpreted as a configuration item. Usage of recipes, classes, and BSPs offers itself a simple way to document the structure of the system. Furthermore, objects that are created or modified in the company should be added to the VCS. Generated packages and binaries are not added to the VCS. Automatic backup process for the generated packages and binaries should be taken into use.

During this work, configuration items were identified. These configuration items were gathered by inspecting the embedded software framework, source codes, and other data. It is important to notice that not all of these configuration items will be added to the VCS. Identified configuration items are listed in Table 4. At the same time with identifying configuration items, also baselines were determined. These baselines are introduced later in this Thesis.

Table 4: Identified configuration items

Configuration item	Added to the VCS
Append file	Yes
Boot loader source codes	No
BSP	Self-produced: yes Other: no
Build reports and statistics	No
Class files	Yes
Configuration files	Yes
Documentation	Yes (partially)
Embedded software framework	No
Images	No
Layers	Self-produced: yes Other: no
Linux kernel source codes	No
Recipes	Yes
Package	No
Root file system	No
Source codes for applications	Self-produced: yes Other: no
Tool chains	No

A technical description and a plan for using embedded software framework as the SCM system is introduced in Chapter 6.

5.5 Open source frameworks

This Section introduces the most popular open source embedded software frameworks.

Usage of open source embedded software frameworks offers strengths listed below (Free Electrons, 2011)

- Open source
- No accessory expenses
- Possible to customize framework as much as needed
- Community support

Weaknesses with the open source embedded software frameworks

- Lack of enterprise support
- Requires expertise
- Incomplete documentation

Linux Target Image Builder

Linux target image builder (LTIB) is a tool used to develop and deploy BSPs and to develop Linux images for various target platforms (Free Software Foundation, 2012). Freescale Semiconductor, Inc originally sponsored LTIB, but currently it is maintained by the Savannah Project. Because of the co-operation with the Freescale, BSPs and other configurations for the most of Freescale products can be found for LTIB. LTIB is licensed under the GNU General Public License V2 or later (GPL). LTIB is a command line interface tool supporting configuration and selection of packages, target architectures and other properties from a graphical menu structure (Figure 5.4).

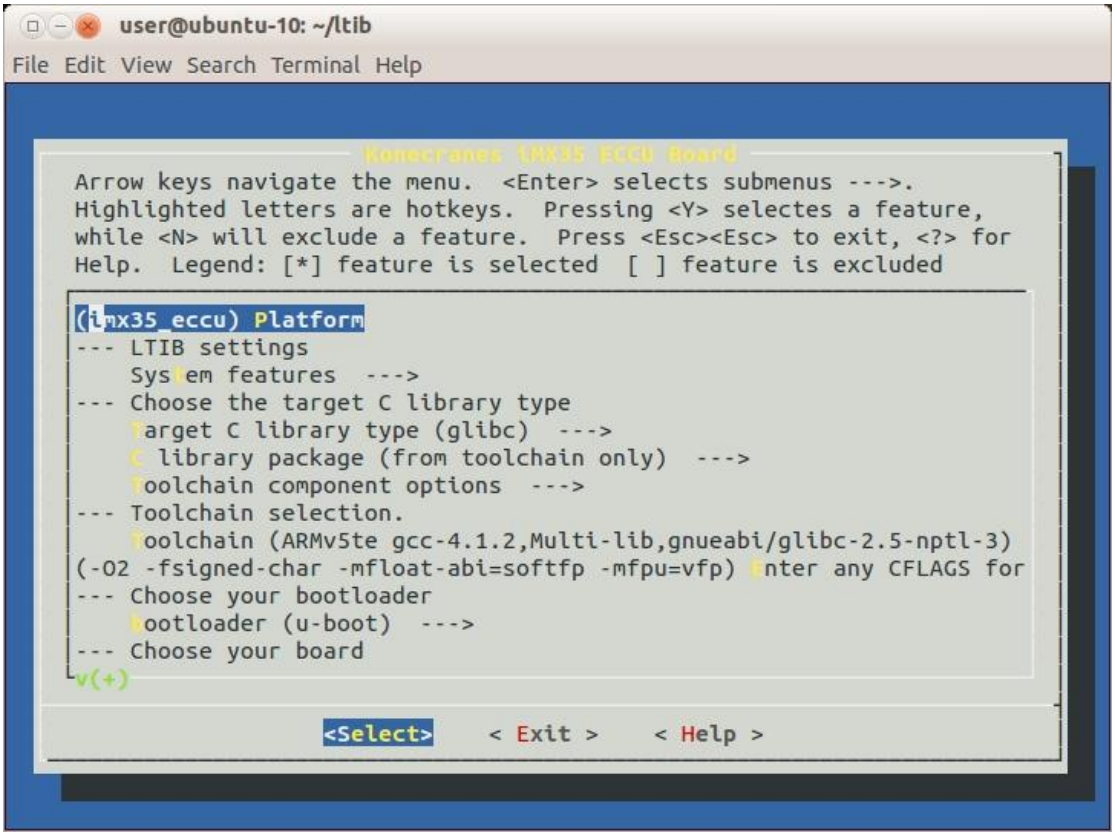

Figure 5.4: LTIB menu

OpenEmbedded

OpenEmbedded is a popular solution for creating Embedded Linux distributions. OpenEmbedded supports many hardware architectures and offers comprehensive amount of releases and tools. OpenEmbedded is easy to customize, which allows multiple other embedded software frameworks to be constructed based on it. OpenEmbedded uses the layer approach. (OpenEmbedded, 2012)

OpenEmbedded offers a lot of useful and powerful features and tools. One example of these tools is BitBake, which is a program used to execute tasks. BitBake is a core component for the OpenEmbedded and it is also used with the Yocto Project. Due to the complexity and the amount of the features, getting familiar with the OpenEmbedded might be a time taking process.

Yocto Project

The Yocto Project describes itself as an open source collaboration project that provides templates, tools, and methods to help developer create custom Linux-based systems for embedded products regardless of the hardware architecture. The Yocto Project, announced in 2010, is a workgroup within the Linux Foundation. The Yocto Project includes recipes for the core system components used with the OpenEmbedded. (Yocto Project, 2012a)

The Yocto Project uses Poky as a platform builder. Poky offers tools to design, develop, build, debug, and test the platform under development (Poky, 2012). In addition, the Yocto Project provides several tools to aid the development process. Application Development Toolkit (ADT) is used to do development of applications for the target platform. The Eclipse plugin enables controlling of the ADT and tool chain from the Eclipse. Hob (Figure 5.5) is a tool that offers a graphical interface to make modifications to configuration and build options.

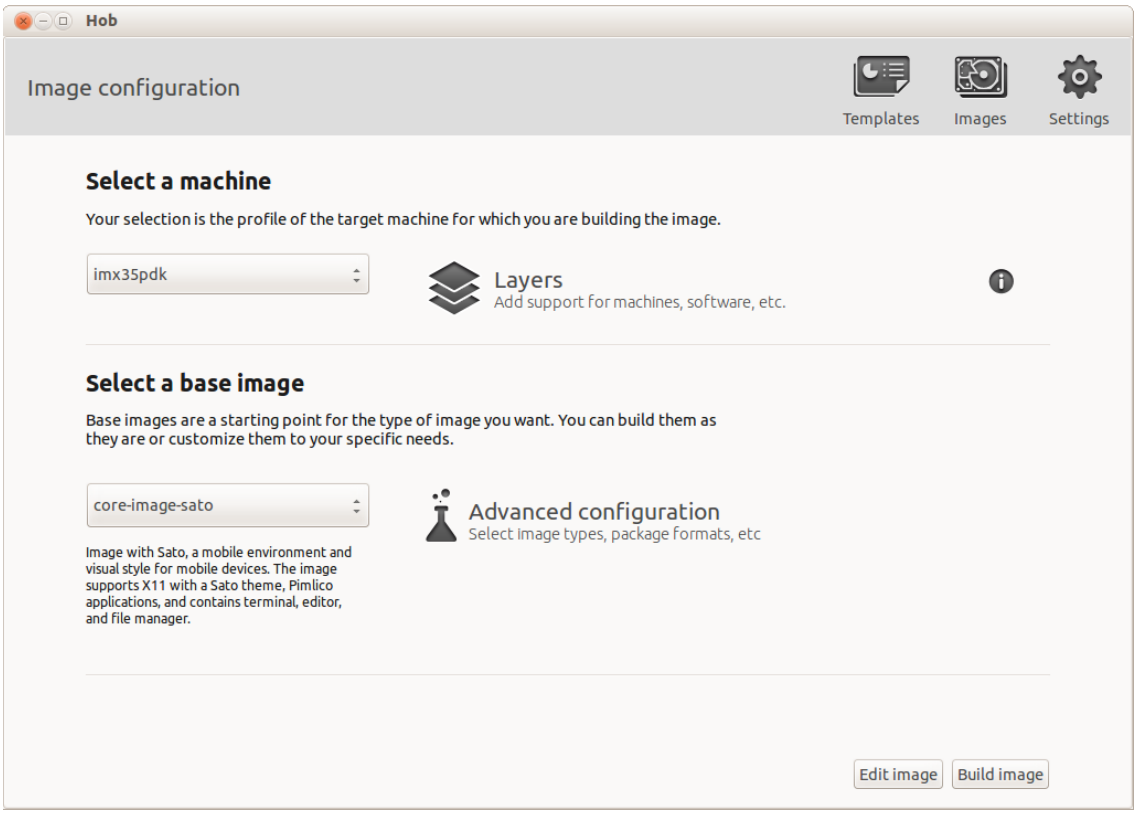

Figure 5.5: Hob

Some commercial embedded software frameworks, like products from Wind River and MontaVista are based on the Yocto Project. For this reason, when using the Yocto Project it is possible to switch using commercial frameworks without major problems. Major companies (Intel, Freescale, Texas Instruments, and Huawei) are supporting the Yocto Project. (Nohau, 2012)

Buildroot

Buildroot is used to generate an embedded Linux system by using makefiles and patches. Like OpenEmbedded and the Yocto Project, Buildroot generates a cross-compilation tool chain, root file system, kernel image, and boot loader image with automated build process. Buildroot supports multiple architectures and provides a simple textual menu interface for setting configurations. Interface for the Buildroot resembles LTIB's interface. Buildroot is licensed under the GNU General Public License V2 or later (GPL). (Buildroot, 2012)

5.6 Commercial frameworks

This Section introduces the most popular commercial embedded software frameworks. Although the following frameworks are commercial products, it is important to notice that commercial does not necessarily stand for proprietary (Free Electrons, 2011).

Usage of commercial frameworks offers strengths listed below (Free Electrons, 2011)

- Graphical development tools
- Automated build, kernel and file system generation
- Linux kernel and tools are well supported and tested
- Very large list of supported target platforms
- Ready solution with Linux kernels, tool chains and utilities
- Better support than with open source frameworks
    - Long term support available (vendor dependent)
- Tools available on multiple operating systems

Weaknesses with the commercial frameworks

- Costs
- Configurability
- Lack of community support

MontaVista

MontaVista Software is a company that provides commercial solutions for developing embedded Linux systems and software. MontaVista offers many tools for different purposes. MontaVista Linux is a distribution that includes Linux kernel, software libraries, and other applications. Instead of using BSPs and recipes to enable the developer to create his own distribution, MontaVista Linux is a ready distribution for a specific target platform. MontaVista Linux is built by using the Yocto Project. (Montavista, 2012).

MontaVista also provides an IDE called DevRocket that includes a group of Eclipse plugins used to help application and system development with the MontaVista Linux. MontaVista also contributes for the Linux kernel and open source communities.

Wind River

Wind River is also a company that provides commercial solutions for developing embedded Linux systems and software. Wind River has a similar solution as MontaVista by providing a ready distribution for a specific target platform. This solution is called Wind River Linux. In addition, Wind River also provides distributions with Android and VxWorks. Wind River offers various tools for the development of embedded systems. Wind River is a founding member of the Yocto Project and Wind River Linux is compatible with it. In addition, Intel Corporation owns Wind River. (Wind River, 2012)

TimeSys

TimeSys is a company that provides services and tools for Embedded Linux development. TimeSys provide a tool called LinuxLink for building an Embedded Linux. TimeSys offers two versions of its tool, LinuxLink FREE and LinuxLink PRO (Timesys, 2012). The first is a web-based tool, which provides an easy approach to build a custom system for the target platform. LinuxLink PRO also provides the same features but with capabilities for customization. LinuxLink PRO also includes tools for other embedded development tasks.

5.7 Comparison of frameworks

Embedded software frameworks have different properties, functions and ways to perform the build process. To support decision-making, a comparison between the most popular embedded software frameworks was carried out in this Thesis. Table 3 describes embedded software frameworks and their technical and lifetime issues compared with each other. All properties are evaluated and marked with colors (green, yellow, red) from good to bad.

The purpose of this comparison is to help to evaluate different embedded software frameworks. Evaluation properties are selected by the writer of this Thesis within the context of the PCCP. For example, when implementing and maintaining the PCCP, support for different architectures plays an important role in choosing an embedded software framework to be used. Viability, documentation, frequent updates, and the general continuity with the tools are also important properties with a long life cycle product like the PCCP.

The evaluation process started with selecting embedded software frameworks to be compared. Both commercial and open source embedded software frameworks were selected to ensure a comprehensive overview of these tools. By studying different options, seven most used and known embedded software frameworks were selected to be included in this evaluation.

Data for this evaluation was gathered by examining web pages of the embedded software frameworks, reading documentations, exploring user experiences from the developer communities, and testing tools. Evaluation properties were selected by using this data and objectives described above.

Table 5: Comparison of embedded software frameworks

Feature/ Framework	Supported architectures	Company or organization founded	Updates	Costs	Documentation	Enterprise support	Developer communities	Partner organizations
LTIB	ARM, PPC, Coldfire	2005	Not actively	Open source	Minimal	Freescale products	Good	Freescale
OpenEmbedded	ARM, PPC, MIPS, x86, x86-64, amd64, avr64	2004	Regularly	Open source	Comprehensive	Independent consultants	Excellent	-
Yocto Project	ARM, PPC, MIPS, x86, x86-64	2010	Stable releases every 6 months	Open source	Comprehensive	Independent consultants	Excellent	Intel, Huawei, Texas Instruments, Wind River, Montavista, etc.
Buildroot	ARM, x86, MIPS, PPC	1999	Stable releases every 3 months	Open source	Good	No	Good	-
MontaVista Linux 6	ARM, PPC, MIPS, x86, x86-64, amd64	1999	Regularly	Commercial	Comprehensive	Yes	Minor	-
Wind River Linux 5	ARM, Intel, MIPS, PPC	1981	Monthly	Commercial	Comprehensive	Yes	Minor	Freescale, ARM, Intel, Texas Instruments, MIPS, etc.
TimeSys LinuxLink PRO	ARM, PPC, Intel, MIPS, Atmel	1995	Regularly	Commercial	Comprehensive	Yes	Minor	-

5.8 Selecting the framework

The Yocto Project was selected to this work based on comparison of the embedded software frameworks. As seen from the Table 5, the Yocto Project is evaluated as the best option for the PCCP. The Yocto Project offers great opportunities today, but also in the future. Arguments to support this selection are listed below

- The Yocto Project operates under Linux Foundation
    - Well organized
    - Support in future
- The Yocto Project is used by the major commercial vendors in the field (MontaVista and Wind River)
    - If needed in the future, allows easy transition from the Yocto Project to commercial products
- Large and active developer base
- Large variety of supported hardware platforms
    - BSP for the current processor (Freescale i.MX35) used in the PCCP available (Github, 2013)
- Comprehensive documentation
- Increasing amount of utility programs
- No costs

Possible problems with this selection

- Starting with the Yocto Project requires a lot of work
- Open source tool
    - Full enterprise support not available

6 LIFE CYCLE MANAGEMENT IN PRACTICE

6.1 Introduction

This Chapter introduces created processes and guidelines for the implementation of the life cycle management with the PCCP. These processes are composed by using the Yocto Project as an embedded software framework. Processes are designed to be as automated as possible to maximize the productivity and to decrease the amount of time used to configure and maintain these systems. Documentation of the Yocto Project (Yocto Project, 2012a) was used as a reference to compose code examples.

PCCP is used for several purposes demanding distinct software. Therefore, controlling of these software products is important during the life cycle of the PCCP. Software products are updated, fixed, and modified constantly. The reason for this is the need for fixing detected defects and adding new features to the software. Typically, defects can be discovered at any time and new features are added to the software periodically, for example twice a year. This Chapter describes how the life cycle management and the SCM is planned to be realized with the Yocto Project and supporting tools.

Processes, code examples, and workflows created and introduced during this Chapter are related to the SCM. Figure 6.1 illustrates how the Yocto Project and the tools used with that are aligned with the SCM implementation phases. Methods described in this Chapter are related to the technical issues within the context of the SCM. Requirements for the SCM are referenced to illustrate how those are planned to be realized by using the Yocto Project and other tools. Furthermore, this Chapter can be seen as a guide for realizing the ALM.

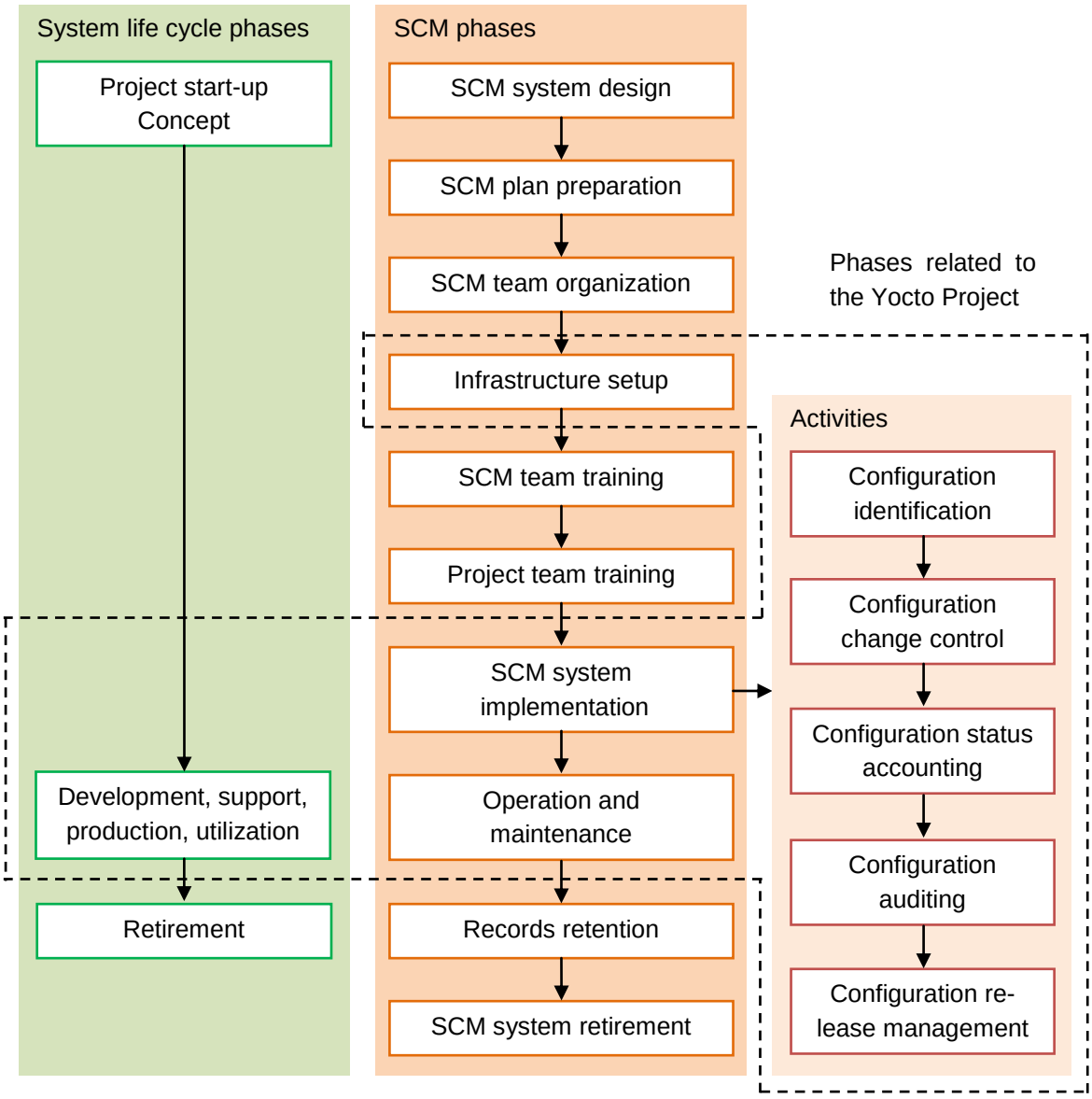

Figure 6.1: SCM implementation phases related to the Yocto Project

6.2 Working environment

Working environment for the development work is designed based on the requirements for the SCM and life cycle management. The designed environment is described in Figure 6.2. It includes working stations for the developers, application designers, and test designers. It also includes the automated build and test server, the local mirror, the version control system (SVN), the issue and bug-tracking tool (Jira), and connections between them. In addition, connections to the internet are allowed.

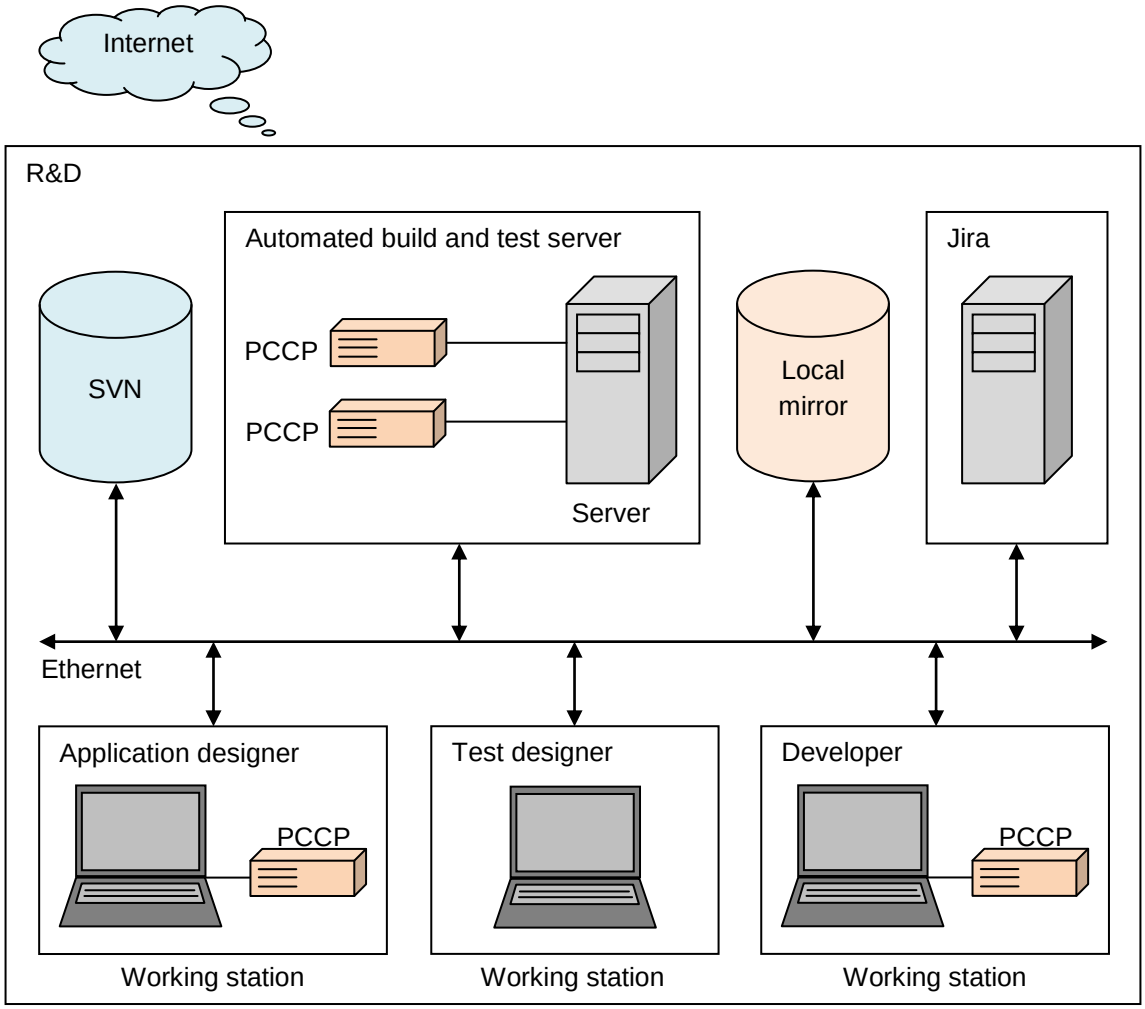

Figure 6.2: Working environment

The working station for the developer, application designer, and test designer is constructed from the following parts

- Desktop computer or laptop
    - Windows operating system (organizational reasons)
    - Ubuntu Desktop Linux distribution running as a virtual machine on top of Windows
        - Yocto Project
            - Hob (Graphical UI)
        - Subversion (SVN) tools
        - Eclipse
            - Yocto Project –plugin
            - SVN plugin
            - Jira (issues and bug tracking tool) plugin
        - Debugging tools
- PCCP connected to the computer (optional)

The automated build and test server is constructed from the following parts

- Server computer
    - Windows operating system
    - Ubuntu Server Linux distribution running as a virtual machine on top of Windows
        - Yocto Project
        - SVN tools
        - Robot Framework (test automation framework)
        - Jenkins (continuous integration tool)
- Several PCCPs connected to the server

Virtual machines are chosen to be used, as they provide an easy way to distribute ready to use development environment to the developers. Other major advantage with the viris the ability to perform backup perform backuping efficiently. Virtual machine image including development environment can be easily copied and later restored and taken into use if needed. Usage of the virtual machines provides a way to fulfill the SCM requirement for reproducibility. Furthermore, usage of the virtual machines allows better compliance with future hardware and software. Linux is chosen as an OS for the virtual machine because the target is running Embedded Linux. Ubuntu was selected out of many Linux distributions because of the following reasons: previous expertise in the organization, comprehensive support by the communities and developers, wide user base, and frequent updates.

VCS contains all necessary files demanding versioning: recipes, configuration files, and source codes for the own software implementations. Configuration items placed to the VCS were defined in Table 4. Version control system is set up locally or purchased as a service from other vendor. In this case, an existing system (SVN) provided by other vendor is used.

The automated build and test server is implemented to execute build and test processes. These processes are executed automatically by the server or manually by the developer. Application designer can also use this environment to compile applications and test functionalities of those applications with previously defined test cases. The server can include a simple web user interface for controlling the server and its functions. In this work, the server is specified to include a web interface for the continuous integration tool (Jenkins).

The local mirror is used to store built packages for all releases, which consists of applications and images for Linux kernels, boot loaders, and root file systems. In addition, source codes for the Linux kernel and boot loader, and downloaded source packages are located at the local mirror. The Yocto Project, installed in the automated build and test server or developer’s computer, downloads packages and source codes from the local

mirror (and from the VCS) instead of using internet as a primary source. Limiting access only to the local mirror and version control system provides more controllable build process, better management of the source code, third party components to be controlled better, and more secure development environment. Usage of the local mirror also prevents accidental inclusion of unwanted proprietary software into the built system. The local mirror is located in a standalone computer or it is integrated as a part of the automated build and test server. In this work, the latter approach is used. Centralization of all packages, external source codes, and images makes backuping more straightforward. Features related to the local mirror are used to fulfill the SCM requirements for reproducibility, safety, and control.

6.3 Software releases

After the software product changes, a new release is created. With the PCCP, software releases will be divided into major and minor software releases. SCM and controlling of the releases are related to each other. The management of the software releases is a part of the SCM activity called configuration release management.

Release schedule for the major software releases is planned in advance and usually the time between releases is fixed (for example 3 or 6 months). The final schedule for the major releases with the PCCP is decided later. Releasing the software updates less frequently simplifies the version management. With the PCCP, software products may remain unchanged for a long time. However, bugs and defects in the software might require immediate updates and releases. In this case, minor releases are distributed when needed.

Figure 6.3 describes a plan for carrying out software product releases with the PCCP. With this method, software is divided into three specific products: Product A, Product B, and Product C. These products have multiple releases: old versions, current production version, and development version(s). In terms of SCM, these releases are called baselines. Releases include specific software components. Software product development is a continuous process, so one release is followed by another release throughout the life cycle of the product until it is retired.

After the release of the software product, the process of storing all necessary objects is executed. These objects should be automatically stored after the build process is executed successfully. A script should be created to automate this process. The objects to archive and backup after each release are listed below (Nohau, 2012):

- Built images: Linux kernel, root file system, boot loader
- Packages generated by the Yocto Project
- Version information of the used recipes and images

- SVN revision numbers (or tags) for the source code used to generate current build
- Details about the environment used to build the release
- License information
- Build reports
- Test reports
- Backup from the environment used to generate the release

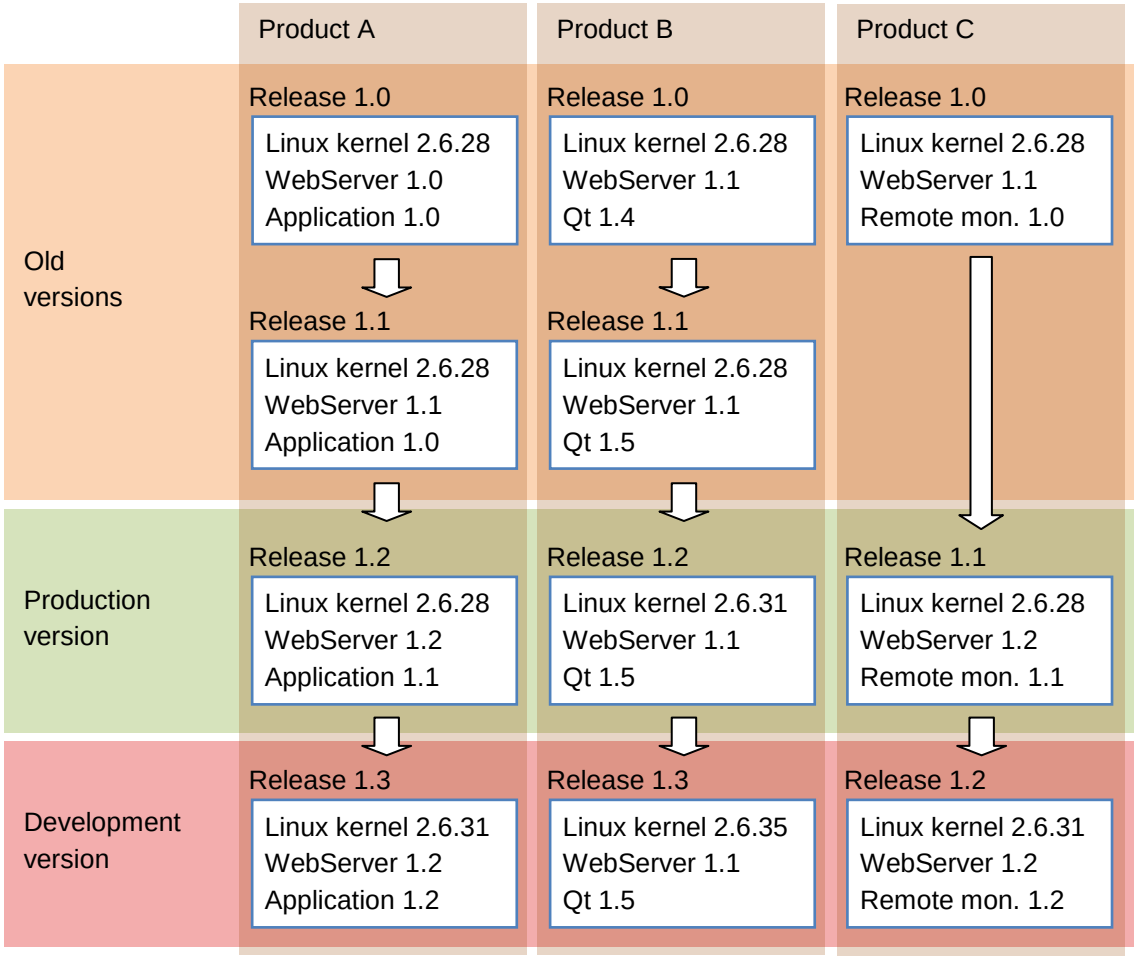

Figure 6.3: Product releases

6.4 Files and directories

The most important file types used with the Yocto Project are introduced in Table 6 Layout of the directory and file structure used on the automated build and test server (or on any computer) is illustrated in Figure 6.4 The introduced directory structure is commonly used with all Yocto Project instances at the company.

Table 6: File types

File	Filename extension	Explanation
Recipe - Image recipe - Baseline recipe - Release recipe - Application recipe	.bb	Build configurations for packages, images, and applications
Class	.bbclass	Recipe class file
Append file	.bbappend	Append file for the recipe
Configuration file	.conf	Global configurations

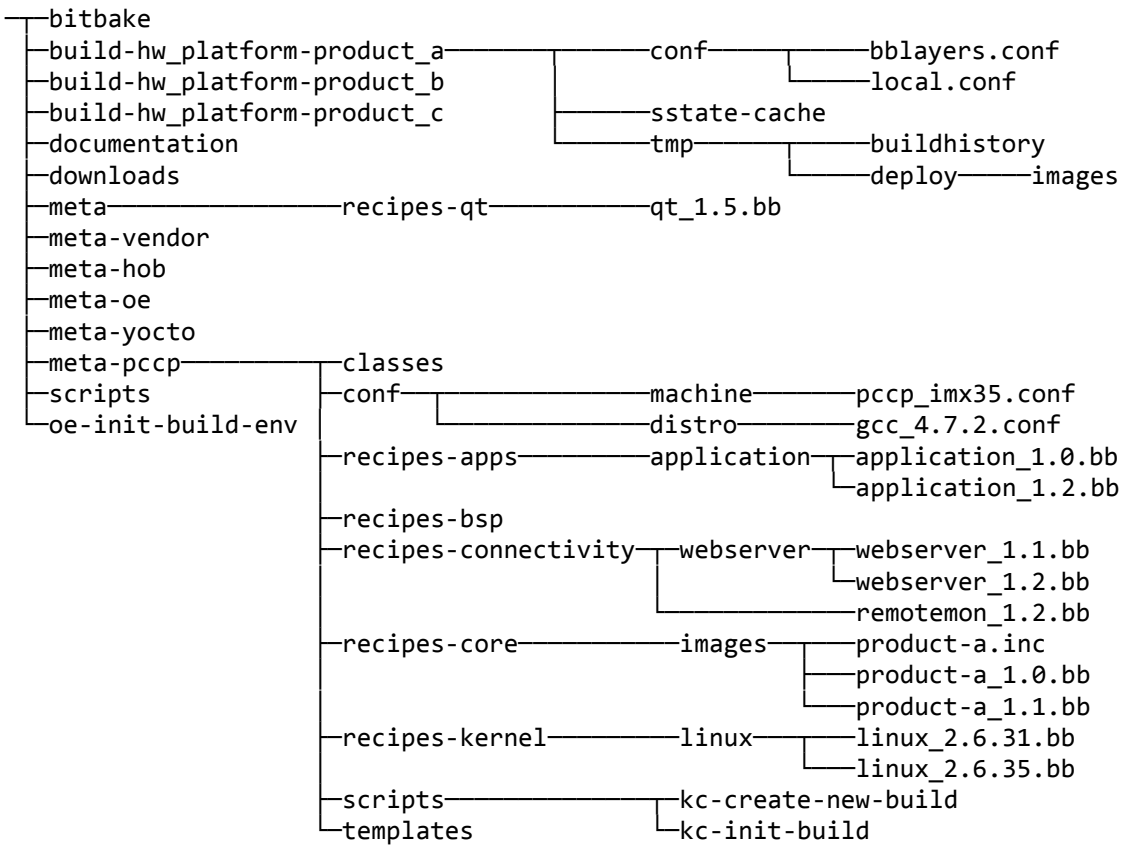

Figure 6.4: Directory and file structure for the Yocto Project

Table 7 introduces configuration files related to the Yocto Project. These files are important when defining the hardware platform and making settings for the build environment.

In theory, changing the hardware platform requires only one simple modification to the configuration file (*local.conf*). However, differences between the hardware platforms might require creating new or modifying existing application recipes to support both hardware platforms.

Table 7: Configuration files

Filename	Explanation
bblayers.conf	Layer configuration file, which determines layers used with the specific build.
pccp_imx35.conf	Configuration file for the hardware platform. The layer can contain several configuration files for the different PCCP variations.
gcc_4.7.2.conf	Configuration file for the distro. This file determines policies for the distro. A layer can contain several distro configurations.
local.conf	Build environment configuration file. Each build directory includes this configuration file. Build-specific hardware platform and distro are defined in this file.

The build directory contains all configuration files and files produced during the build process. With the PCCP, separate build directories must be created for each software product. This division allows product builds to be more controllable. After the build directory is created and configured, configuration files located at the build directory should not be modified afterwards. New build directory is created by using the script *kc-create-new-build*. This script was created to automate the creation of a new build directory and it sets default configurations for the build directory. A simple rule is that one build directory contains builds only for one hardware platform and one software product. Directory *build-hw_platform-product_a* is the build directory for the Product A. Usage of the script for creating a new build directory is illustrated in code (1). Build directory also includes directories for the build statistics (*buildhistory*) and built images (*images*).

```
$ source kc-create-new-build build-hw_platform-product_a   (1)
```

All retrieved source packages are located at the *downloads*-directory and it is configured to be shared between all build directories. Sharing this directory decreases the disk usage at developer's computer.

The directory structure includes multiple layers that are either created by the developer or retrieved from elsewhere. Directories *meta*, *meta-hob*, *meta-oe*, and *meta-yocto* are layers related to the Yocto Project. Directory *meta-vendor* is a layer that determines BSPs, recipes, images, and configuration files for a specific board or processor architecture. The *meta-vendor* layer is implemented by the processor manufacturer or by the developer community. The layer described above is useful when starting to realize a layer (directory *meta-pccp*) for the PCCP. The *meta-pccp* layer is implemented to support specific needs of the PCCP. Furthermore, this layer includes the hardware platform configurations (BSPs), distro configurations, Linux kernel recipes, boot loader recipes, application recipes, image recipes, and other configurations for the PCCP. Although,

recipes and classes can be inherited from the other layers, the *meta-pccp* layer is considered as an independent layer. This layer was created only for the purposes of the PCCP. Directory structure used in the *meta-pccp* layer is common to all layers. In addition, s*cripts*- and *templates*-directories are added to support more controllable usage and to aid in starting to use the Yocto Project. All of the layers described above are defined as configuration items.

The *meta-pccp* layer is the only directory under this directory structure, which is added to (or retrieved from) the SVN. The reason for this is that append-files provide an easy way to customize recipe files in other layers. This way, there is no need to modify original recipes from other layers, which would require placing those other layers under the SVN. In addition, recipes and classes can be inherited from the other layers. With these features, all changes to recipes to support the PCCP are added to the *meta-pccp* layer. This makes installation process of the build environment much easier, since other layers are in an original form and available on the Internet. These methods provide an efficient way to manage traceability and share changes on the applications with other developers.

Recipes are located at each layer in directories named with a prefix *recipes*. These directories are named to describe a specific purpose. The layout of a recipe file is illustrated at Appendix A, Code 3.

6.5 Reproducibility, traceability, scalability, and control

Extra attention is required with the SCM implementation for software products with requirements for reproducibility, traceability, and control. During the life cycle of embedded software, it is probable that in some point it is necessary to access old versions of the software. Embedded software frameworks, together with the VCS and continuous integration tools, provide methods to realize reproducibility, traceability, and control. This Section describes how the Yocto Project is configured by using specific code lines. In addition, some configurations can be made using graphical user interface (Hob). The reason for these code examples is to illustrate how the requirements for the SCM, reproducibility, traceability, scalability, and control, are realized using the Yocto Project.

Methods described below are selected to be used to support specific features of the PCCP. These features are multiproduct usage, long life cycle, configurability, and scalability. Usage of these selected methods provides a way to implement an efficient and controllable system.

The hardware platform and the distro for the specific product are assigned by setting code (2) to the file *local.conf* under the product's build directory. With this method, different hardware platforms and tool chains can be supported. This method is an approach to provide scalability with the Yocto Project.

```
MACHINE ?= “(hw_platform)”                                    (2)
DISTRO ?= “platform”
```

Versioning of the recipes, images, and releases must be used to enable software traceability. Code (3) illustrates how versioning is applied with recipes. This variable represents the version of the recipe and it is added to all recipes. The initial version for the recipe is *r0*. After changes occur to the recipe, version number is increased. This method is used together with the VCS to allow changes to the recipes to be traceable.

```
PR = “r2”                                                     (3)
```

Code (4) illustrates versioning of the application. This variable represents the version of the application or release. In addition, the version number is used in the file name of the recipe. For example, in this case the file name for the recipe would be *application_1.3.bb*. Version numbering is used to support different versions of the application source code. This method provides an efficient way to control different versions of an application. It also enables traceability between the different versions of the application.

```
PV = “1.3”                                                    (4)
```

Effective usage of the VCS plays an important role in life cycle management. The Yocto Project provides mechanisms to retrieve source codes from all the most commonly used VCSs. Retrieving source code from the repository during the build process allows latest updates to be included into the build. Furthermore, the source code can be retrieved from the repository by using a revision number. This approach is very useful when building an older version of the software or the newest version of the software is not stable. Retrieving source codes from the VCS with a specific revision number is a way to enable reproducibility. Usage of the SVN as a source code location with a specific revision number is illustrated in code (5). Correspondingly, code (6) illustrates how the latest revision of the source code is retrieved.

```
SRCREV = “2492”                                               (5)
SRC_URI = “svn://svn.server.com/platform/yocto;protocol=http;
           module=source_directory”

SRCREV_pn-${PN} ?= "${AUTOREV}"                               (6)
PV = "1.0+svnr${SRCPV}"
SRC_URI = “svn://svn.server.com/platform/yocto;protocol=http;
           module=source_directory”
```

The methods described above provide a useful way to control different versions of the recipes. Guidelines for the versioning of the application recipes were defined during this

Thesis. These guidelines are following. Multiple recipes for one application can be created with different version numbers. Older recipe versions retrieve source codes from the repository with a specific revision number, in other words those versions are *frozen* and not to be modified. These recipe versions are used to build the product delivered to the customer. The latest version of the recipe is in a development use and the recipe automatically retrieves the latest source code available. Usage of these guidelines provides control between the application versions.

As mentioned earlier, all source codes produced in the company are located at the SVN, which means that all those source codes are retrieved from the SVN. Since the PCCP is in a prototyping stage, all recipes created are configured to retrieve the latest revision of the source code from the SVN.

The purpose of the first software development activity was to determine requirements for the software product. Tracing requirements to the actual software is a useful feature when testing and evaluating the product. Requirements are described by using an issue and bug-tracking tool called Jira. With this tool, a unique key value is assigned for every requirement. This key value can be for example *req-23*. Specific requirement is referred to from the recipe by a variable (7). The original purpose of this variable is to categorize recipes. With the *meta-pccp* layer, this variable is defined to specify requirements for the recipes. Graphical user interface and shell interface can be used to categorize recipes according to this variable. The variable provides a mechanism to include or exclude recipes with specific requirement. With this method, traceability from the requirements to the application is ensured.

```
SECTION = “req-23”                                    (7)
```

Another way to categorize recipes or packages based on the requirements is to use package groups. In other words, recipes and packages dependent on a specific requirement can be easily included or excluded from the image by using package groups. A package group is used to integrate multiple recipes into single group, which can then be referred to. Package group also provides a method to include multiple applications to the image by referring to a single package. With the *meta-pccp* layer, package groups are used to group recipes to provide a way to include or exclude recipes easily. For example, applications used for a wireless communication are added to one package group.

License information is required for all recipes. If the recipe is not marked to use any license, the build process cannot be executed for that recipe. Usage of the license information in the recipes provides control and safety. This feature is illustrated in code (8).

```
LICENSE = “GPLv2”                                                    (8)
LIC_FILES_CHKSUM = “file://COPYING;
            md5=kas82me0l15kv5mtqzx3kt42g4l6op32mw”
```

Sometimes it is necessary to make modifications to the Linux kernel, boot loader, or to some other externally retrieved source code. However, to preserve traceability, modifications should not be made directly to the source code. The Yocto Project uses patches to make modifications to the source code. Code (9) illustrates adding a patch to a recipe. Patches are located at the *recipe*-directories. Simple rule is that all changes to the external source code must be applied by using patches.

```
SRC_URI = “file://fix_some_bug.patch \                               (9)
           file://add_new_feature.patch”
```

In an environment where multiple developers are doing development work, building an image from scratch is not usually recommended. The Yocto Project provides a feature called *shared state cache,* which allows built packages to be shared between different computers. Usage of the shared state cache located on the automated test and build server accelerates the build process on the developer’s machine, since only packages that have changed needs to be built. Build time is significantly decreased by utilizing this feature. Code (10) illustrates how location for the shared state cache is determined in the build configuration file (*local.conf*). This variable is automatically set when creating a new build directory with script *kc-create-new-build*.

```
SSTATE_MIRRORS ?= “\                                                (10)
  file://.* http://company.com/platform/sstate/PATH \n \
  file://.* ftp://company.com/platform/sstate/PATH \n \
  file://.* file:///local/directory/sstate/PATH”
```

All downloaded source packages, and source codes for the Linux kernel and boot loader are located at the local mirror. The Yocto Project is defined to use these locations by adding code (11) to the distro configuration file. In addition, other variables are included in the distro configuration file to make the Yocto Project more controllable. These variables are illustrated in code (12). Usage of these variables in the recipes allows system locations to be changed easily.

```
MIRRORS =+ "\                                                       (11)
ftp://.*/.* http://local_mirror_ip/platform/sources/ \n \
http://.*/.* http://local_mirror_ip/platform/sources/ \n \
https://.*/.* http://local_mirror_ip/platform/sources/ \n"
```

```
PCCP_SVN_MIRROR = “svn.company.com/pccp”                    (12)
PCCP_MIRROR = “ftp://<server_ip>/packages”
PCCP_SSTATE_MIRROR = “ftp://<server_ip>/sstate-cache”
PCCP_DOWNLOAD_MIRROR = “ftp://<server_ip>/downloads”
```

6.6 Images for the software releases

The PCCP is used for several purposes and all of those purposes can require different applications. Management of these different software products is carried out by using image recipes. These image recipes are normal recipes, which are used to determine what to include in the image. SCM requirement for reproducibility is realized with the image recipes. In addition, image recipes provide means to produce identical images at any time. The Yocto Project provides various ready to use image recipes, but often it is necessary to create own image recipes to satisfy requirements for the current project and software.

Figure 6.5 illustrates the process of creating or modifying image recipes with the Yocto Project. This process is performed by modifying image recipe files directly, using web user interface, or using graphical user interface. During this Thesis, image recipes were created by writing image recipe files directly.

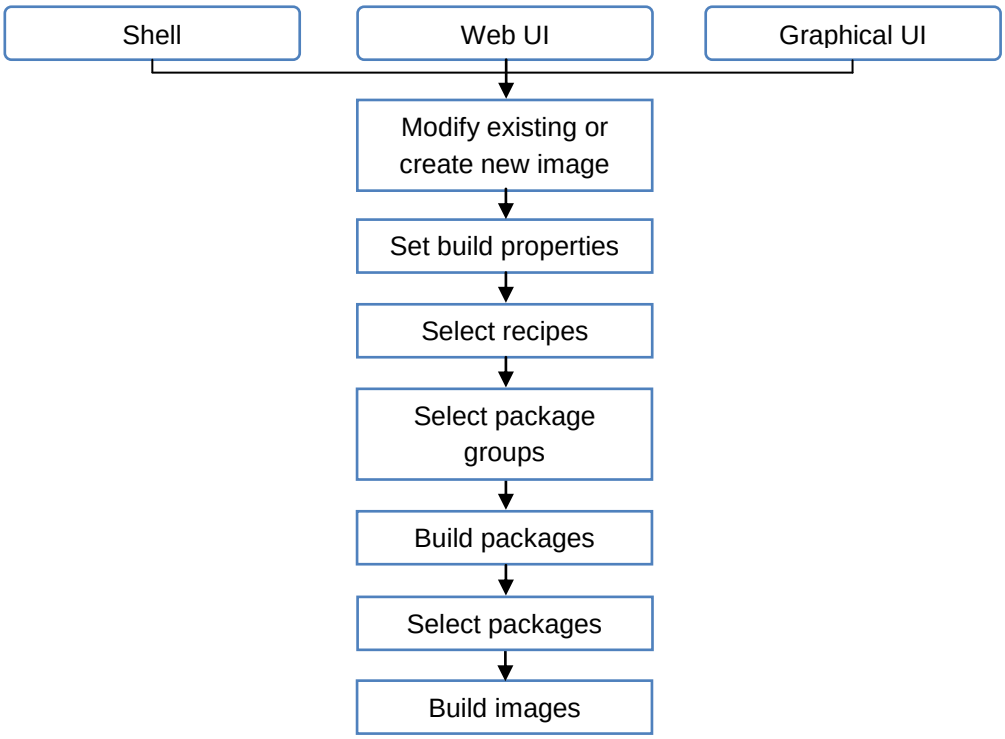

Figure 6.5: Creating and modifying images

Figure 6.6 illustrates image recipes within the context of the products and releases introduced earlier. To support different products with various versions, concepts of an image baseline and image release recipes are created in this Thesis. Every software

product has an image baseline recipe, which determines basic configurations for the software product. All products also have additional image release recipes, which determine properties for the specific releases. These properties include information about the included recipes and versions for those. Image release recipe inherits the contents of the image baseline recipe. Furthermore, image release recipes can contain information about recipes to be excluded from the image due to application of some license. Image release recipes are named by adding the name of the product and the version of the release, for example *product-a_1.2.bb*. Image baseline and image release recipes must be used to ensure reproducibility of the images. In addition, usage of these recipes provides a way to increase the scalability of the embedded software framework. Image baseline and image release recipes are located at the *meta-pccp* layer. Examples for the image baseline and image release recipes are illustrated in Appendix A, Code 2 and Code 4. Other image recipes can also be created if needed. For example, image for debugging or testing new functionalities, or a minimal image, which includes only basic components needed to operate the PCCP.

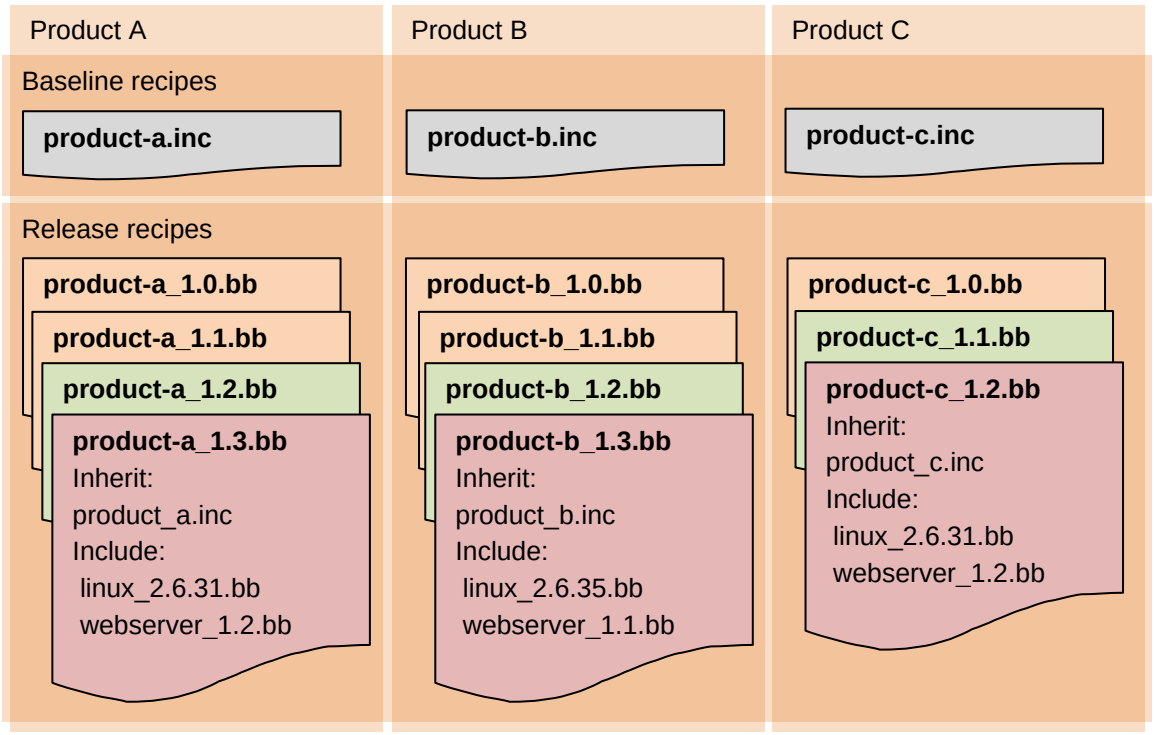

Figure 6.6: Image recipes for products

Command (13) illustrates initialization of a previously created build directory. A custom initialization script (*kc-init-build*) was created to support specific needs of the PCCP. Usage of the image release recipes allows individual release builds to be categorized under each product's build directory. After the command (13) is executed, image for the product is built by using the selected image release recipe. Starting the build process at command line is executed with command (14), which builds the release image by using the latest version of the image release recipe. Older versions of the image release recipes can be built by adding code (15) to the *local.conf*.

```
$ source kc-init-build build-platform-product_b          (13)
$ bitbake product_b                                      (14)
PREFERRED_VERSION_product-b = "1.1"                      (15)
```

The build process creates images with informative file names. This file name includes product name, release number, name of the PCCP, build time, and the contents of the image. For example, file name for the image created using the image release recipe called *product-a_1.0.bb* is *product_a-1.0-pccp-20121214100544.rootfs.tar.bz2*.

Packages built using recipes are included in the image by adding a variable (16) to the image recipe. This variable is located at the image baseline recipe or at the image release recipe depending on how generally current feature is used. Usage of this variable provides an efficient and clear way to control which features to include in the image.

```
IMAGE_INSTALL = “webserver_1.2 application_1.2”          (16)
```

Sometimes it is necessary to exclude an application that is using a specific license from the image. Adding code (17) to the image recipe excludes applications using the specified license.

```
INCOMPATIBLE_LICENSE = “GPLv3”                           (17)
```

Disabling recipes from the image is possible by adding code (18) to the image recipe.

```
BBMASK = “.*/meta-yocto-bsp/recipes-graphics/”           (18)
BBMASK .= “|recipe”
```

During this Thesis, an image recipe for the minimal PCCP configuration and an image recipe for a configuration including all applications used in the company were created. In addition, two image recipes for the purpose of application development were created. These images include tools for debugging, performance testing, and other purposes.

6.7 Workflow

In this Thesis, a workflow for the development work with the PCCP is designed. The workflow is illustrated in Figure 6.7. The workflow describes the behavior of the automated build and test server. In addition, the workflow is automated as much as possible. Depending on the current configuration of the server, execution of the build process is manual, automatic, or scheduled. Manual build process is used when the developer executes the build process by hand. If the build process is configured to be automatic, all software is built when a commit is made to the SVN. Scheduled build process is executed in fixed periods, for example once a week. Automatic and scheduled build

processes can be combined. Usage of the automatic build process increases the stability (SCM requirement) of the build system enabling problems to be discovered as soon as possible.

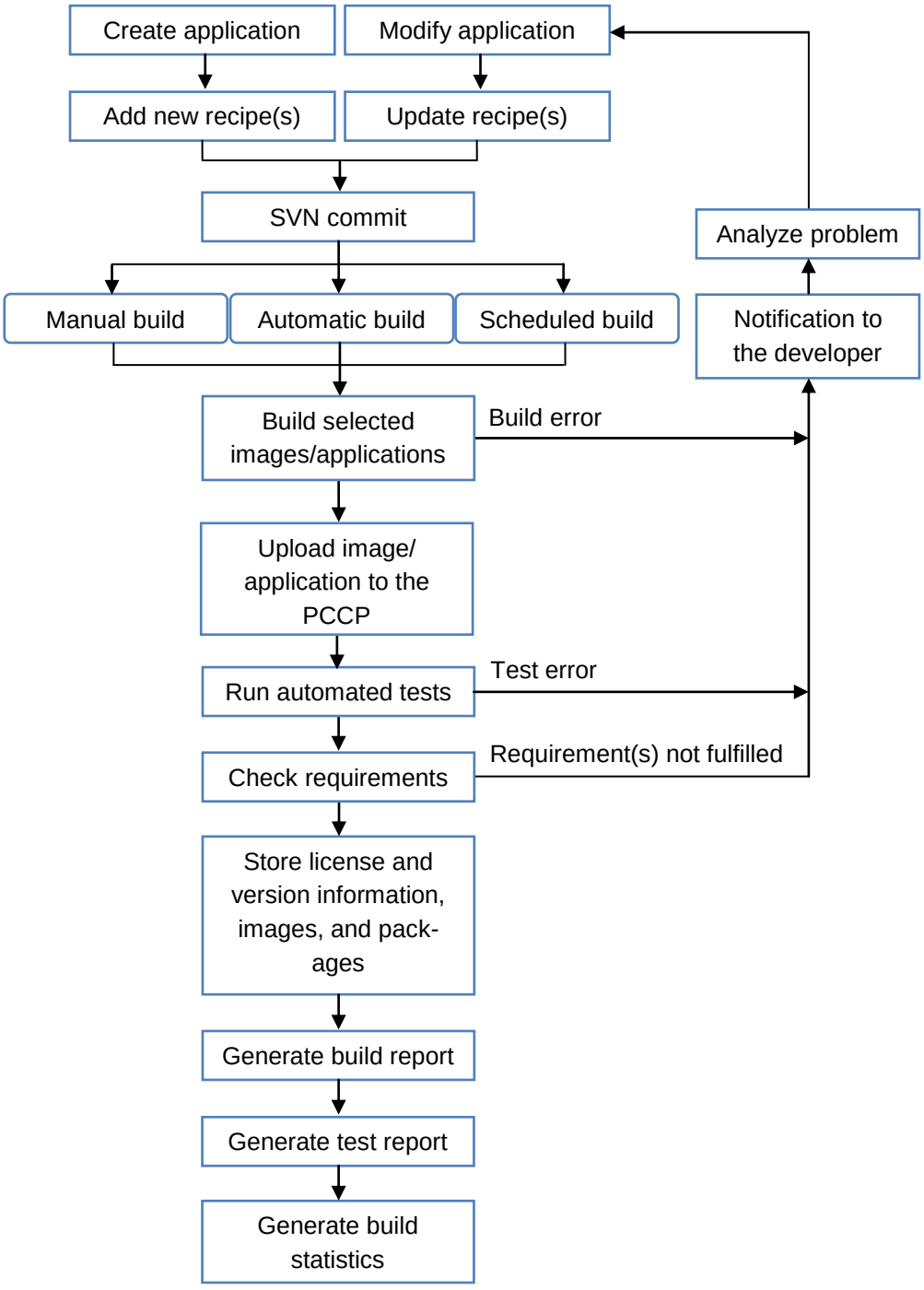

Figure 6.7: Workflow

Building the entire software product is not productive if only a single application needs to be compiled and tested. In practice, applications will be built using the automatic build process. Build process for the images is scheduled to be executed once a week. Currently, all build process methods described above are used with the PCCP.

6.8 Documentation

Recipes are a good place for documentation. Code (19) describes what information must be included in all recipes. Methods introduced in Section 6.5 also provide positive value for the documentation. Adding documentation to the recipes makes auditability easier and makes recipes much more readable and understandable.

```
DESCRIPTION = “Description of the recipe”                    (19)
SUMMARY = “Summary for the packaging systems”
AUTHOR = “Creator of the recipe”
HOMEPAGE = “Web site where recipe information is found”
```

Documentation for the built images must be created. Yocto Project provides a feature called build history to create this documentation automatically. Adding code (20) to the file *local.conf* enables creation of the build history for the built image. The build history includes information about the built image and provides an option to commit this information automatically to GIT. Features described above provide a way to fulfill requirements for traceability by allowing contents of the image to be examined later. The build history is also a way to increase reproducibility and traceability. The reason for this is that changes made to the image recipes can be found easily by viewing the build history.

```
INHERIT += “buildhistory”                                    (20)
BUILDHISTORY_COMMIT = “1”
```

The build history can be viewed from the command line by examining text files or by accessing a web interface. The build history also creates graphs that describe dependencies between the built packages and it is an efficient tool for tracing dependencies between the packages. The dependency graph is illustrated in Figure 6.8. For example, when a bug is discovered or a build error occurs, dependencies between the packages are needed to be traced to find the actual source of the issue. The build history is automatically enabled when creating a new build directory with the script *kc-create-new-build*.

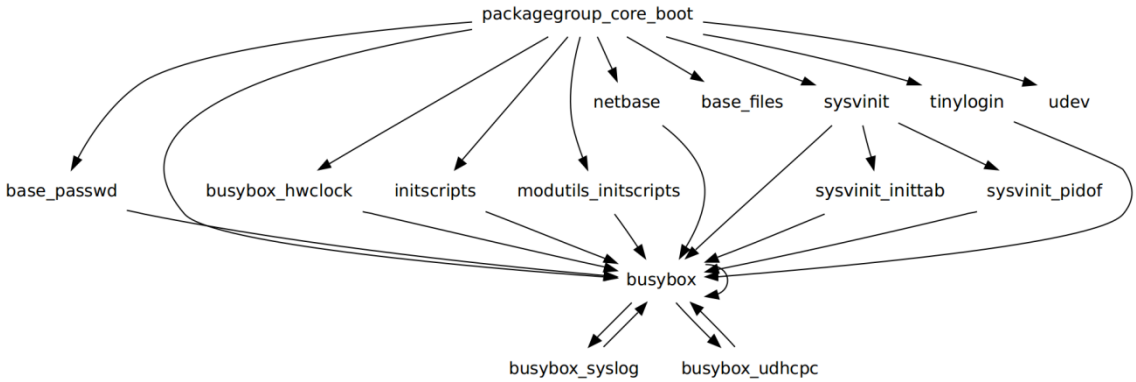

Figure 6.8: Dependency graph

6.9 Application development

Code (21) illustrates how the recipe is identified to support specific hardware platforms. The application cannot be built for other hardware platforms than specified into this variable. This feature is useful if the recipe is not working or it is not tested with all hardware platforms. Usage of this variable in all recipes allows managing of supported hardware platforms and enables scalability of the system.

```
COMPATIBLE_MACHINE = “(hw_platform|other_platform)”        (21)
```

The embedded software framework is not necessarily needed for compiling applications for the PCCP. Applications can be compiled by using a standalone tool chain. However, embedded software framework is used to generate the standalone tool chain. The Yocto Project generates a tool chain installer with command (22). The tool chain included in the installer is the tool chain used with the build directory where the command (22) is executed. This installer provides a simple way to distribute tool chains for the application developers.

```
$ bitbake meta-toolchain                                   (22)
```

As an outcome for this project, application designers are offered two different methods to execute build process for the application. The methods are called server build and local build. These methods are created to support different ways to do development work. The methods support multiple operating systems. In addition, there is no special need to install development tools to the developer’s computer. These methods are illustrated as a workflow in Figure 6.9.

With the server build, the application designer does not have to install any other development tools than SVN on the computer. The application designer can choose any text editor to write and modify source code. Automated build and test server builds the application after the application designer has committed application source codes to the SVN. Furthermore, the server can execute automatic test cases for the built application. After the server has processed all assigned tasks, it notifies the application designer with an email if something went wrong. Built packages for the application are available at the local mirror after the build process is executed.

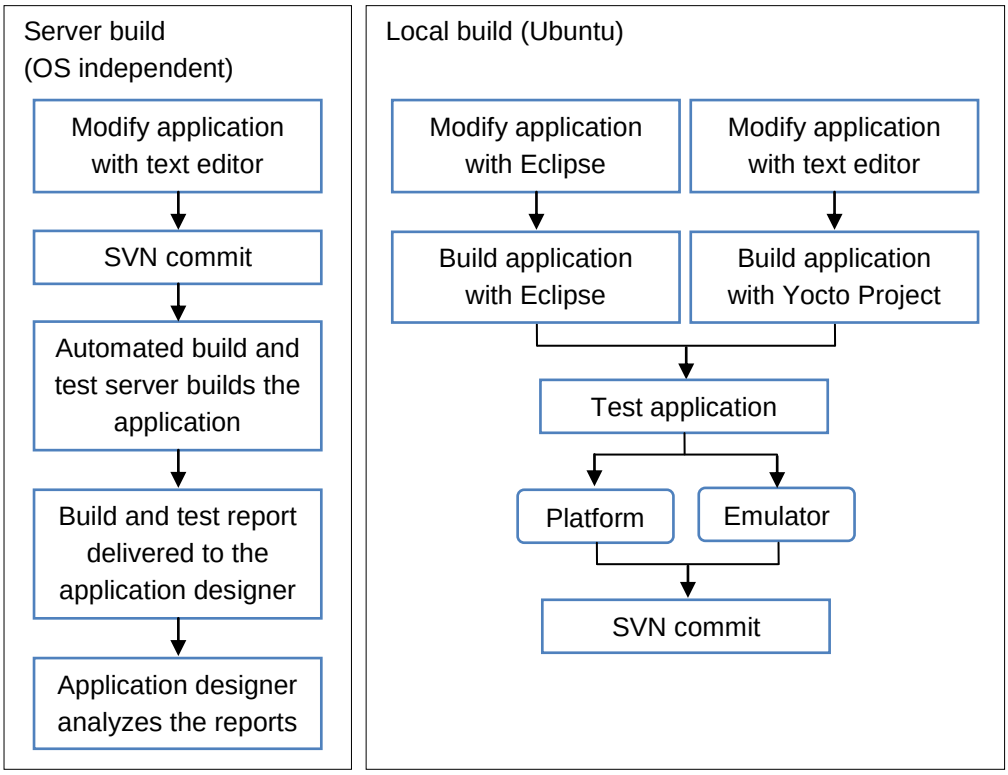

Figure 6.9: Application development

With the local build, the application designer has a working station described in Section 6.2. Local build process is performed by using Ubuntu (running as a virtual machine). This virtual machine is available as a ready to use package for the application designers. With tools installed in this virtual machine, application designer can realize all development tasks locally. The application designer writes and modifies source code using text editor or Eclipse. The Yocto Project plugin installed in Eclipse provides an efficient and simple way to execute the build process and create new recipes directly from Eclipse. In addition, plugins for the Jira and the SVN are installed in Eclipse. Testing and debugging application is done with the actual target PCCP or with an emulator by using tools supported by Eclipse. As mentioned earlier, two image recipes were created to support application development. With these images, Eclipse can be used for remote deployment and debugging of the application running at the actual target PCCP. With all of these methods, Eclipse can be used to realize all tasks related to the application development for the PCCP. When doing development work with a text editor instead of Eclipse, build process for a single application is executed by command (23).

```
$ bitbake remotemon                                    (23)
```

Installers for the tool chains used with the PCCP are available at the local mirror. The application designer can create their own customized working station by installing these tool chains.

7 CASE: DEPLOYMENT OF THE YOCTO PROJECT

7.1 Background

As mentioned earlier, all practical work done in this Thesis is related to the PCCP. Figure 2.3 illustrates the hardware platform of the PCCP. Along with the PCCP, Freescale's i.MX35 Product Development Kit (PDK) (Freescale Semiconductor, Inc, 2012a) was used. It provided an easy approach for testing embedded software frameworks, since the BSP for this PDK is available for the Yocto Project. Similarities between this PDK and the PCCP also help creation of a BSP for the PCCP.

Previously, LTIB was used as an embedded software framework for the PCCP. However, because of the limitations and problems with the LTIB, some other embedded software framework had to be taken into consideration. Multiple products with requirements for reproducibility, traceability, and control could not have been handled with the LTIB.

7.2 Objectives

The objective for this case was to deploy an embedded software framework for the PCCP and install additional tools to support development work. The Yocto Project was chosen as an embedded software framework for this case. PCCP specific source codes for the Linux kernel and boot loader is obtained from the LTIB. The main purpose for this case was to configure the Yocto Project so that the output images would be similar to what LTIB creates. In addition, features described in Chapter 6 will be added to the system deployed.

7.3 Configuring the Yocto Project

7.3.1 Setting up the Yocto Project

Installing the Yocto Project on a computer is a straightforward process. The most important thing to consider at this point is to figure out what version of the layers should be retrieved. Support for processors and applications vary between different versions. Usually, recommended layer releases for the processor are described in the instructions of the processor's BSP layer. In this case, all layers are retrieved using the same release

to maximize compatibility. Steps executed to install and configure the Yocto Project are described below. Embedded software framework is ready to be used after these steps.

1) Retrieve the Yocto Project by using GIT (release: danny)
2) Retrieve other necessary layers by using GIT
    a. Layer for the Freescale's ARM platforms *(meta-fsl-arm*, release: danny)
    b. Layer for the OpenEmbedded (*meta-oe*, release: danny)
3) Create a new layer for the PCCP (*meta-pccp*)
    a. This is described in detail in Section 7.3.2
4) Create a build directory for the PCCP (*build_pccp*)
5) Configure build properties for the created build directory
    a. Set used layers
    b. Select machine
    c. Set build options (parallel compilation, number of threads)
    d. Set a local download directory
    e. Set a local shared state cache directory
    f. Select distro
    g. Enable build history recording
    h. Set mirror locations for package downloading and shared state directory

Commands needed to execute steps 1 and 2 are illustrated in Appendix B. A script was written for automating tasks listed under the step five. This script automatically sets default configurations for new build directory. The developer can then make modifications to these if the default configuration is not suitable. Usage of this script also helps to get started when the developer is unfamiliar with the Yocto Project. In addition, a script to initialize a previously created build directory was written. These scripts are located at the *meta-pccp* layer.

7.3.2 Creating a new layer

A new layer was created to support the specific needs of the PCCP. Starting point for the creation of the *meta-pccp* layer was the *meta-fsl-arm* layer, which provides BSPs for the Freescale's ARM processors. Usage of the *meta-fsl-arm* layer with the Yocto Project was first tested with the PDK, which provided a good way to learn using the Yocto Project.

The actual implementation of the *meta-pccp* layer started after testing with the PDK. The first step to start the creation of an own layer was to create a machine configuration file (*pccp_imx35.conf*) for the PCCP. This file determines used processor type (ARM11), preferred versions of the Linux kernel and boot loader, settings for serial console communication and boot loader, additional machine features, and file types for the images.

The next step was to add support for the PCCP specific Linux kernel (2.6.31) and boot loader (U-Boot 2009.08). Custom source codes for these were available, so only thing needed was to create recipes for the kernel and boot loader. In addition, default configuration files were added for both of them. Building an actual working system for the target PCCP was done after these recipes were implemented. At this point, images for Linux kernel, boot loader, and root file system were loaded to the PCCP and tested to be functioning properly.

Adding applications and other configurations started after the base for the layer was implemented. Table 8 describes the distro configuration files created for the layer. Distro configurations are divided into several files, which provide a way to inherit common configurations in multiple distros. These distro configuration files (*.inc*) provide company specific configurations such as addresses for the PCCP related systems. In this case, only one distro (*gcc-4.7.2.conf*) was created.

Table 8: Created files related to the distro configuration

File	Description
gcc-4.7.2.conf	Distro configuration file supporting tool chain gcc 4.7.2.
distro-common.inc	Configuration file including all common configuration settings. This file is inherited by the distro configuration file.
distro-mirrors.inc	Configuration file including company specific locations for the SVN and local mirror. This file is inherited by the distro configuration file.
distro-premirrors.inc	Configuration file including the Yocto Project specific locations for the upstream packages. This file is inherited by the distro configuration file.

Table 9 describes image recipes created during this case. Four different image recipes were created. These image recipes support development work with the PCCP. At this point, image baseline or image release recipes were not created. This is because the PCCP is still in the prototyping stage and the final decisions about the products where it will be used are yet to be made.

Table 10 describes scripts and templates created for the *meta-pccp* layer. The purpose of these templates is to support more efficient development work and to help developers, unfamiliar with the Yocto Project, to produce recipes more easily. Scripts were created to support the creation and initialization of a build directory. These scripts decrease the workload because developer does not have to set configurations by hand every time when the build directory is created or initialized.

Table 9: Created image recipes

File	Description
pccp-minimal.bb	Determines a minimal image, which includes only the basic functionalities needed to use the PCCP. Target PCCP using this image is only accessible through serial console.
pccp-minimal-sdk.bb	Same image as the *pccp-minimal.bb* but also including tools to support debugging and performance evaluation.
pccp-ssh.bb	Determines an image, which includes all basic applications used with the PCCP. Target PCCP using this image is accessible through serial console and SSH.
pccp-ssh-sdk.bb	Same image as the *pccp-ssh.bb* but also including tools to support debugging and performance evaluation.

Table 10: Created scripts and templates

File	Description
kc-create-new-build	A script that creates a new build directory. The script automatically sets default configurations for the build directory.
kc-init-build	A script that initializes build environment for the specific build directory.
template-autotools_1.0.bb	A template recipe for an application, which use autotools for compiling.
template-initscript_1.0.bb	A template recipe for adding an init script for the embedded Linux.
template-makefile_1.0.bb	A template recipe for an application, which uses makefiles for compiling.
template-bblayers.conf	A template configuration file for the default layers.
template-dev_local.conf	A template for *local.conf*. This file is used as a default configuration file for the local working station.
template-server_local.conf	A template for *local.conf*. This file is used as a default configuration file for the automated build and test server.

Busybox (collection of UNIX tools) configuration file was modified to include necessary features like network utilities. In addition, recipes for the Linux init scripts were added to support initialization of the hardware and file systems during the system startup. One of the biggest tasks was to add recipes for the company specific applications. These applications support special features like remote monitoring and machine controlling. Applications were included in the image *pccp-ssh*.

All drivers needed by the hardware platform were added to the *meta-pccp* layer by creating recipes for them. These drivers add support for the connectivity and platform features. Adding recipes for these drivers were tricky because drivers needed to be compiled with the tool chain used with the Yocto Project. This tool chain was much newer than the tool chain previously used to compile these drivers. For this reason, it was important to test these compiled drivers thoroughly. In addition, it was not an easy task to learn using the Yocto Project for compiling external source codes using make-files. Other smaller things included to the *meta-pccp* layer were a recipe for adding user accounts for the Embedded Linux and configuring an ftp-server to the image *pccp-ssh*.

The last step at the layer creation process was to test all features with a real target. All features, except one application related to the machine control, were tested.

Figure 7.1 illustrates dependencies for the *meta-pccp* layer. The layer becomes dependent on the other layer if a recipe inherits another recipe or class from another layer. In addition, the layer becomes dependent if a recipe is appended (recipe append file) from another layer. In practice, these dependencies mean that all of these layers must be installed if the developer is using the *meta-pccp* layer

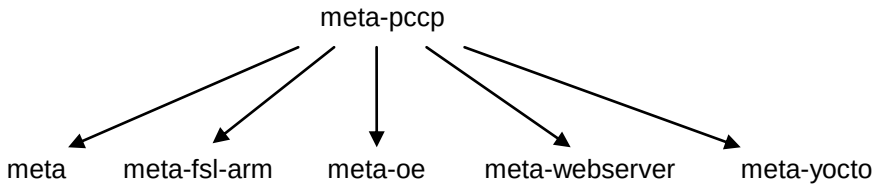

Figure 7.1: Layer dependencies

7.4 Virtual machine for the working station

7.4.1 General structure

Figure 7.2 illustrates the structure of the created virtual machine for the working station. This virtual machine is distributed for the application designers (and other developers). As this virtual machine and all tools are installed and configured, the virtual machine only needs to be copied to the developer's computer. After that, only making configurations for personal settings, like SVN username and password, is needed.

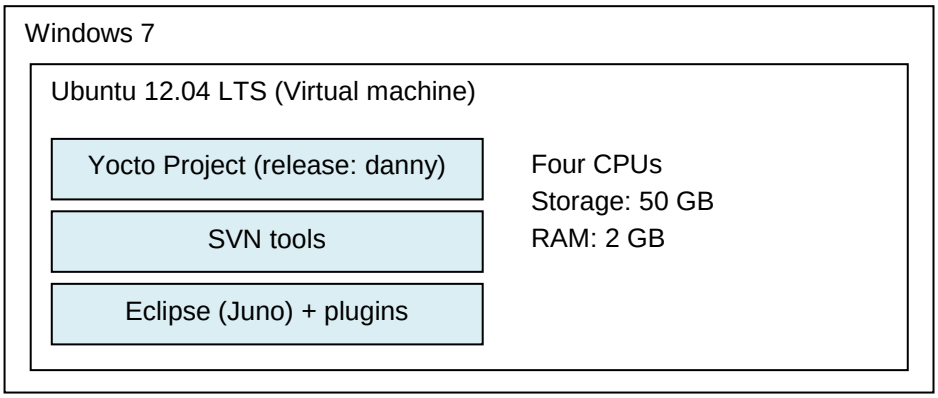

Figure 7.2: Virtual machine for the working station

7.4.2 Installing OS

Ubuntu version 12.04 (precise) was selected to be used as an OS for the virtual machine. This version of Ubuntu was selected because the Yocto Project is supporting it. In addition, this version is called a long-term support version, which guarantees support for the OS for 5 years after the release.

There are a few things to be considered when creating the virtual machine. First thing is to figure out how much storage space is needed. Requirement for space is depending on the images built with the Yocto Project. With this in mind, size of the data storage was specified to be 50 Gigabytes. Another thing to consider is how to configure network interfaces to provide connections to the Internet and target PCCP. In addition, serial communication needs to be assigned to allow the target PCCP to be connected through serial cable.

In addition, tools and programs needed by the Yocto Project were installed on the virtual machine. User account (*user*) for the OS was created for development purposes.

7.4.3 Yocto Project

The Yocto Project was installed on the virtual machine by following the steps introduced in Section 7.3.1. The Yocto Project was installed under the home directory of *user*. After the Yocto Project was installed and configured, all available images at the *meta-pccp* layer were built. This decreases developer's workload, because developer does not have to build all images from scratch.

The *meta-pccp* layer is retrieved from the SVN during the Yocto Project installation. After the installation, SVN username and password are deleted from the virtual machine. Developer needs to provide individual SVN credentials when updating the *meta-pccp* layer at the first time after taking the virtual machine into use. Furthermore, the developer should remember to update and make commits to ensure that the latest version of the *meta-pccp* layer is available for other developers.

In addition, Application Development Toolkit (ADT) for the Yocto Project was installed. This tool provides a plugin for the Eclipse, an emulator (Qemu) to simulate the target PCCP, and other development tools.

7.4.4 Eclipse

The actual development work is performed by using Eclipse. This work includes tasks, such as source code writing, creating and modifying recipes, and debugging applications with the target PCCP or on an emulator.

Plugins were installed in Eclipse to support systems and tools used for the SCM. Subclipse-plugin provides a way to access SVN from Eclipse. Atlassian Connector for Eclipse allows adding comments and making changes to issues located at Jira. The Yocto Project is controlled from Eclipse by the plugin installed with ADT. All development work related to the PCCP can be done by using Eclipse with plugins described above.

7.5 Automated build and test server

7.5.1 General structure

Figure 7.3 illustrates the general structure of the created automated build and test server. Ubuntu is running as a virtual machine. This is because the computer running this virtual machine is also running other virtual machines. User accounts for the server were created for all developers doing development for the PCCP. In addition, an user account for administrative purposes was created.

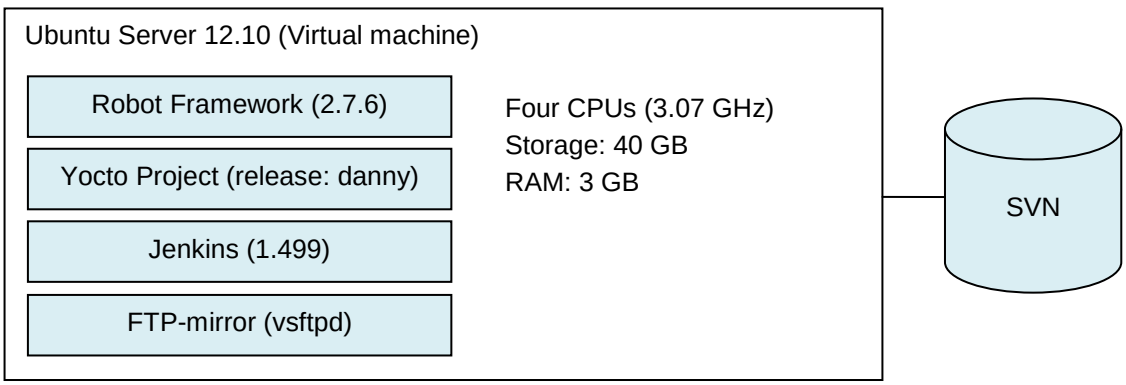

Figure 7.3: Automated build and test server

In future, one or multiple PCCPs should be connected to the automated build and test server. This connection provides a way to execute test cases with real target PCCPs instead of running simulated tests.

7.5.2 Jenkins

Jenkins was installed to perform tasks related to the continuous integration. Jenkins is accessible to all developers, inside the company network, through a web browser. The developer can create jobs that can be configured to build applications, images, or anything else. Jenkins is configured to show basic information about the jobs for everyone. However, only persons who have logged in can make modifications to the jobs and execute the build process. Jenkins offers multiple ways to control who can log in to the system. In this case, Jenkins is configured to allow users who have user account for the automated build and test server to also log in to the Jenkins with the same credentials. Individual jobs are created in Jenkins for the images implemented in the *meta-pccp* layer. In addition, jobs for the most important applications were created.

Jenkins allows usage of shell commands to execute external builds. This is the way that the Yocto Project is used with the Jenkins. Jenkins is configured to build all images once a week. In addition, application builds are executed every night if the *meta-pccp* layer or any source codes under the SVN have changed. If some errors occur during the build process, email including the build log will be sent to the developer(s) specified in the job settings.

Jenkins supports making automatic tests after the build process is executed. These tests are executed by using the Robot Framework. In this work, this feature was briefly tested to be working, but not implemented all the way. Other useful add-on for the Jenkins is the Jira plugin. This plugin allows comments to be added automatically to the correct task in Jira. In addition, it allows changing the progress of the issue.

7.5.3 FTP mirror

The local mirror described in Figure 6.2 is included in the automated build and test server. The local mirror is realized by adding a File Transfer Protocol (FTP) mirror. This mirror is accessible through a web browser or an FTP client. Anyone can download files located at the FTP mirror, but no one is allowed to make any modifications. The FTP mirror provides a simple way to share images built using the automated build and test server, tool chains, guides, and other files.

Figure 7.4 illustrates the directory structure on the FTP mirror. *Deploy*, *downloads*, and *sstate-cache* -directories are used by the Yocto Project installed on the automated build and test server. These directories actually contain the same files as the corresponding directories under the build directory. In this case, the build directory is configured to use these custom directory paths to allow those to be shared with other Yocto Project instances at other computers. Furthermore, this provides an easy way to distribute built images for developers. The *Packages*-directory contains PCCP specific source codes,

and other files in a packaged format. The *Toolchains*-directory includes all tool chains available to be used with the PCCP.

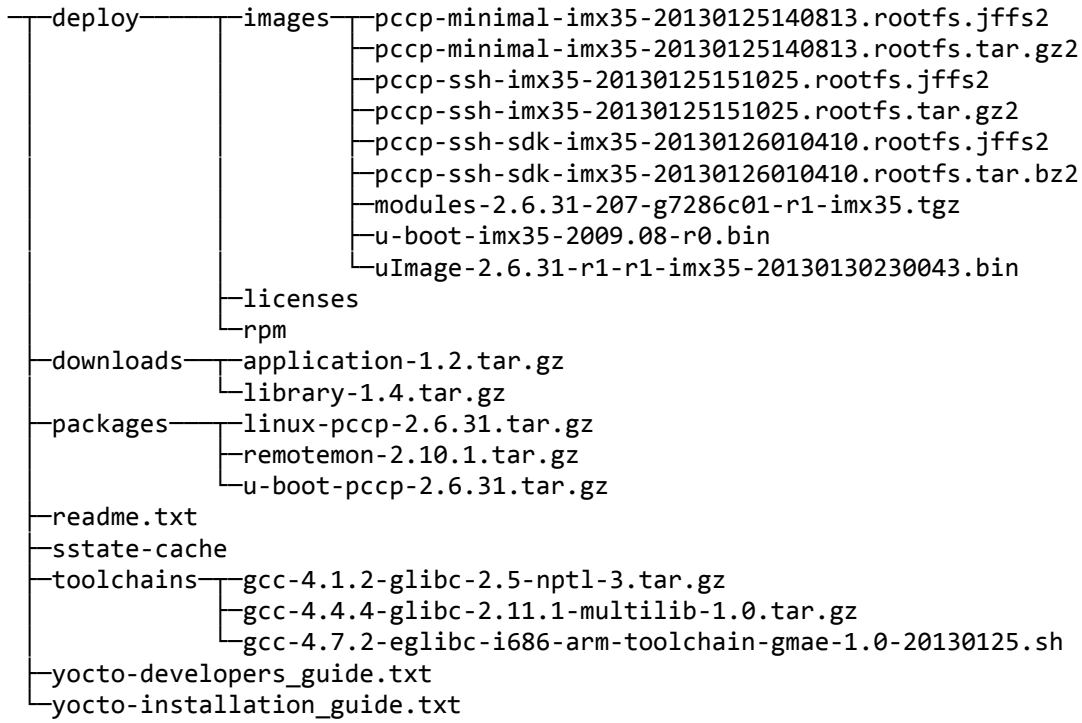

Figure 7.4: Directory and file structure for the FTP mirror

7.6 Process quality

Measurements must be done to ensure the quality and effectiveness of the development process during the life cycle of the PCCP. Standard for the software quality (ISO/IEC 9126, 2003) determines internal and external metrics. In addition, this standard defines several quality models. Table 11 describes metrics created in this Thesis. These metrics belong to the models called usability, maintainability, and portability. These metrics provide a simple way to monitor development process. The main purpose of these measurements is to provide an easy and accurate method to monitor how much development and maintenance work is done. Measurements are also used to monitor the performance and output of the build process. Single numbers measured at one point are not significant. Instead, the main point of these measurements is to follow the trend of the changes in the values.

Table 11: Metrics

Property	Value
Amount of new files added to the *meta-pccp* layer when adding a new application	Files/application
Amount of files modified at the *meta-pccp* layer when making modification to existing application	Files/application
Amount of files added to the *meta-pccp* layer when making modification to existing application	Files/application
Amount of work hours used to add new application to the *meta-pccp* layer	Hours/application
Amount of work hours used to make modifications to existing application	Hours/application
Time used to build an image from scratch	Minutes/image
Time used to build an application from scratch	Minutes/application
Time used to execute test cases for an image	Minutes/image
Disk usage	Gigabytes
Size of the images	Megabytes
Bugs/defects found	Bugs/recipe

7.7 Conclusion

During the implementation of this case, the most important tools needed for the life cycle management and for the SCM were taken into use. These tools offered a way to realize the automated build and test process. With the above results, objectives for this case were fulfilled.

Some problems occurred during the case implementation. The most common reason for these problems was the unfamiliarity with the tools. As the case implementation proceeded and tools became more familiar, the pace of the development work increased. Usage of the PDK as learning and testing platform played an important role. The learning curve would have been too steep without any examples of how to use the Yocto Project with an actual embedded device. Furthermore, one useful aspect discovered during the case implementation was that the developer communities are a good place to search and ask for help.

8 RESULTS AND CONCLUSIONS

8.1 Results of the work

In this Thesis, processes for the life cycle management were created. Technical details and guidelines for these processes were planned and realized. The Yocto Project layer for the PCCP was created, automated build and test server was set up and taken into use, and a virtual machine for the development usage was produced.

Table 12 describes the amount of files produced during the creation of the *meta-pccp* layer. Amount of lines produced and average amount of lines per file were counted to provide statistics to evaluate upcoming workload. Total amount of directories in the *meta-pccp* layer is 53. Table 13 describes the amount of files, which were not produced during this Thesis, but are located at the *meta-pccp* layer.

Workload needed for the maintenance of the *meta-pccp* layer is reduced once the foundation for the *meta-pccp* layer is implemented. During this Thesis, all basic functionalities and applications for the PCCP were added in the *meta-pccp* layer. Afterwards, adding new applications or new functionalities, testing existing applications, and fixing bugs are the biggest workload related to this layer. After application recipe is added to the *meta-pccp* layer, there should be no reason to make modifications to that recipe unless a new feature is added to the application or bug discovered in the recipe.

Table 12: Self-produced files in the meta-pccp layer

File	Number of files	Code lines	Code lines per file
Recipe file	52	1888	36
Append file	5	45	9
Configuration file	5	90	18
Scripts	2	69	35
Template (recipes and configuration files)	6	575	96
Other files	6	347	58
Total	76	3014	40

Table 13: Files retrieved elsewhere but located at the meta-pccp layer

Type	Number of files	Code lines	Code lines per file
Other files	57	11119	176

Work hours used to implement the case is described in Table 14. Counted hours were directly used to create the case. Hours used to study and evaluate used software and systems (Yocto Project, Jenkins, and Eclipse) were not counted. 180 hours were used to implement the case. Directly calculating this would mean that it took roughly a month to implement this case, but in practice, it took about two months. This is because during the case implementation, work hours were spent for studying and testing these issues in general.

Table 14: Work hours used for the case

Task	Work hours
Working station installation	11,5
Yocto Project installation	9
Creating the *meta-pccp* layer	
- General configuration	41
- Machine configuration	3
- Linux kernel recipes	18,5
- Boot loader recipes	3
- Recipes related to the HW platform	15,5
- Application recipes	28,5
- Image recipes	6,5
- Scripts and templates	3,5
Eclipse	15
Setting up the automated test and build server	
- Yocto Project	7
- Jenkins	8
- FTP-mirror	2
Writing guides and instructions	8
Total amount of working hours used for the case	180

Estimations of the workload required to perform different development tasks are composed to Table 15. These estimations are based on the knowledge gathered during this Thesis and the statistics above. These estimations assume that the developer who is doing those particular tasks is familiar with the development process and tools.

Table 15: Workload estimations

Task	Estimated workload
Setting up a working station	1 hour
Adding a new recipe to the *meta-pccp* layer for a simple application	10 min-1 hour
Adding a new recipe to the *meta-pccp* layer for a complex application	1 hours-5 hours
Adding support for Linux kernel to the *meta-pccp* layer (depending highly on used processor and how it is supported with different kernel versions)	10-70 hours
Adding support for a new hardware platform to the *meta-pccp* layer	2-10 hours
Adding a new image recipe to the *meta-pccp* layer	10 min-2 hours
Adding support for a new connectivity, IO, or R&D card to the *meta-pccp* layer	10-60 hours

The workload required for the maintenance of the *meta-pccp* layer will decrease in the future, as developers are getting more familiar with the concepts of the Yocto Project. On the other hand, new features or applications for the PCCP will increase the workload. In addition, new versions of the Yocto Project and other layers are released periodically. Updating the *meta-pccp* layer to support newer versions will require extensive testing before taken into use. The testing can be separated from the actual development work. It is also important to evaluate if there is an actual need to adopt newer versions. However, not every new version should be automatically taken into use. The most important new versions are those, which include major bug fixes or useful features.

The layout of the recipes and other configuration files will not change drastically between the different versions of the Yocto Project. This means that recipes created using older versions should also work with the newer versions. Although, there is no guarantee that dependencies between the layers will not become an issue. These dependencies might be broken if a newer version of a recipe is taken into use.

Figure 8.1 illustrates one possible outcome for the life cycle of the PCCP. This life cycle estimation illustrates possible changes in the hardware platform. There is also estimation on how these changes would affect to the *meta-pccp* layer.

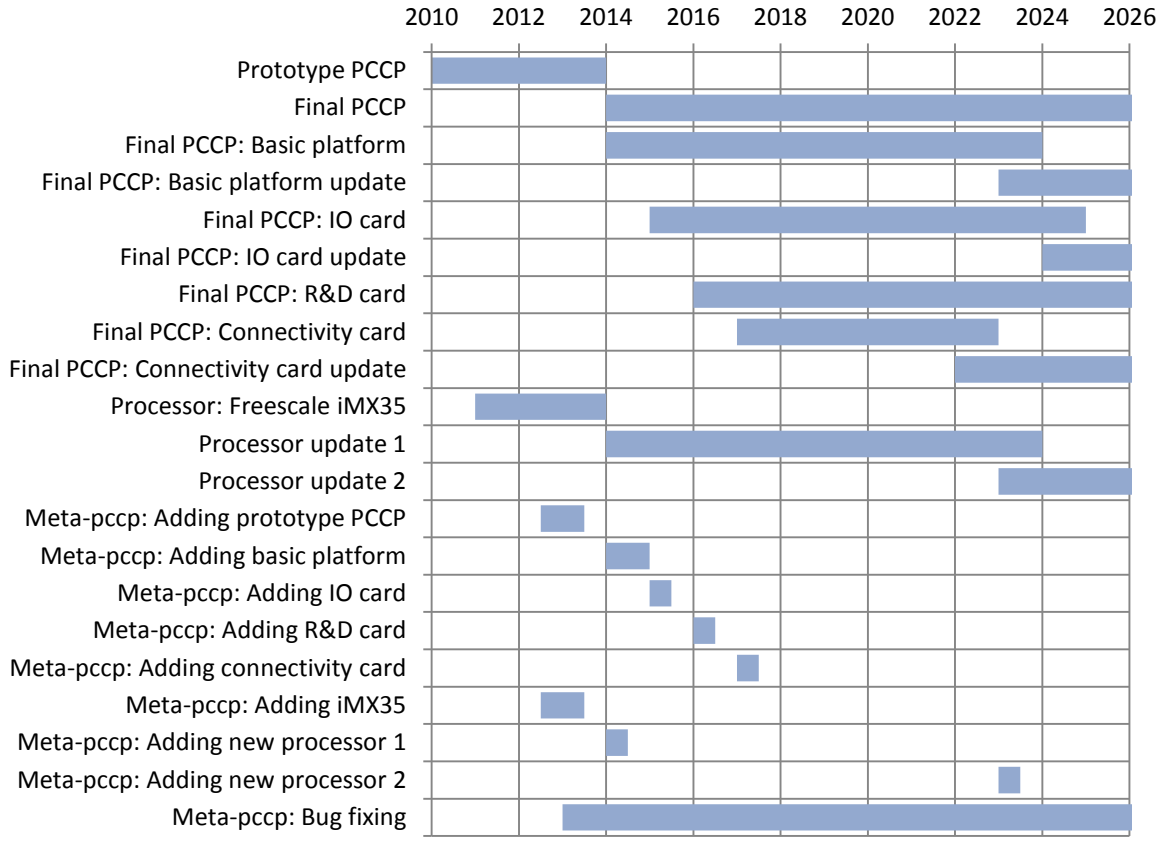

Figure 8.1: Life cycle of the PCCP

Processes created during this work provide a way to automate build and test processes. Modifying and creating new applications is made as easy as possible. In addition, the amount of workload to start doing development work for the PCCP have decreased with processes, tools, and methods introduced in this Thesis. The Yocto Project and other tools with the PCCP specific guidelines enable the requirements for the valuable SCM system to be fulfilled. These processes and tools have been taken into partial use in the company. In addition, the created automated build and test server has been utilized by other projects.

8.2 Future work

This Thesis introduces and defines processes for the life cycle management. These processes were realized during the case implementation. A lot of work is still needed to fulfill all basic requirements for the usable development environment for the PCCP.

There are some technical issues or tasks, which still need to be implemented. One task is to add Robot Framework test cases to Jenkins. These test cases are already created, but the integration in Jenkins needs to be done. Consequently, the automatic test procedure can be executed with the build process. The second task is to add support for two previously used tool chains to the *meta-pccp* layer. The third task is to add support for the newer version of Linux kernel to the *meta-pccp* layer. Another major thing, which

needs further studies and testing, is the Yocto Project's support for other processors. In the near future, it is possible that the processor used with the PCCP changes. Available BSPs for different processors must be researched and evaluated.

During this Thesis, only technical issues related to the SCM (and ALM) were discussed. Additionally, SCM team organization and other SCM properties should be taken into account. Finally, metrics created to measure process quality should be used to ensure that development work is done correctly and efficiently.

REFERENCES

Bailey, Brian, Martin, Grant and Anderson, Thomas. 2005. Taxonomies for the Development and Verification of Digital Systems. s.l. : Springer, 2005. 0-387-24021-7 (Electronic).

Berger, Arnold S. 2002. Embedded Systems Design - An Introduction to Processes, Tools, and Techniques. s.l. : CMP Books, 2002. 1-57820-073-3.

Buildroot. 2012. Buildroot. [Online] November 2, 2012. [Cited: November 2, 2012.] http://buildroot.uclibc.org.

Chappell, David. 2008. What is application lifecycle management? [Online] December 2008. [Cited: October 4, 2012.] http://www.davidchappell.com/WhatIsALM--Chappell.pdf.

Dömer, Rainer, Gertslauer, Andreas and Müller, Wolfgang. 2009. Introduction to Hardware-dependent Software design. Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific. January 19-22, 2009, pp. 290-292.

Ecker, Wolfgang, Müller, Wolfgang and Dömer, Rainer. 2009. Hardware-dependent Software - Principles and Practice. s.l. : Springer, 2009. 978-1-4020-9436-1 (Electronic).

Eclipse Foundation. 2012. Eclipse - The Eclipse Foundation open source community website. [Online] 2012. [Cited: October 5, 2012.] http://www.eclipse.org.

Elemo, Henri. 2008. Defining Software Configuration Management for Product Development. Helsinki University of Technology. 2008. Master's thesis. http://lib.tkk.fi/Dipl/2008/urn012267.pdf.

Free Electrons. 2011. Embedded Linux system development. [Online] February 21, 2011. [Cited: October 8, 2012.] http://free-electrons.com/doc/embedded_linux_sysdev.pdf.

Free Electrons. 2012. Embedded Linux system development. [Online] October 8, 2012. [Cited: October 11, 2012.] http://free-electrons.com/doc/training/embedded-linux/slides.pdf.

Free Electrons. 2010. Introduction to embedded Linux. [Online] May 26, 2010. [Cited: December 4, 2012.] http://free-electrons.com/doc/embedded-linux-introduction.pdf.

Free Software Foundation. 2012. LTIB (Linux Target Image Builder) - Summary [Savannah]. [Online] 2012. [Cited: October 4, 2012.] http://savannah.nongnu.org/projects/ltib.

Freescale Semiconductor, Inc. 2012a. i.MX35 Product Development Kit (PDK) Product Summary Page. [Online] November 1, 2012a. [Cited: November 1, 2012.] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX35PDK.

Freescale Semiconductor, Inc. 2012b. Tower System. [Online] October 11, 2012b. [Cited: October 11, 2012.] http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=0152106740#na.

Github. 2013. Freescale - Github. [Online] March 6, 2013. [Cited: March 6, 2013.] https://github.com/Freescale.

Haikala, Ilkka and Märijärvi, Jukka. 2006. Ohjelmistotuotanto. s.l. : Talentum, 2006. 952-14-0850-2.

IEEE 12207-2008. 2008. ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes . [Online] January 31, 2008. [Cited: October 3, 2012.] http://ieeexplore.ieee.org/servlet/opac?punumber=4475822. 978-0-7381-5664-4 (Electronic).

IEEE 24748-1-2011. 2011. Ieee guide--adoption of iso/iec tr 24748-1:2010 systems and software engineering--life cycle management--part 1: guide for life cycle management. [Online] June 3, 2011. [Cited: October 15, 2012.] 978-0-7381-6603-2 (Electronic).

IEEE 24765-2010. 2010. Systems and software engineering -- Vocabulary. [Online] December 15, 2010. [Cited: Sebtember 24, 2012.] http://ieeexplore.ieee.org/servlet/opac?punumber=5733833. 978-0-7381-6205-8 (Electronic).

IEEE 828-2012. 2012. IEEE Standard for Configuration Management in Systems and Software Engineering. [Online] March 16, 2012. [Cited: Sebtember 24, 2012.] http://ieeexplore.ieee.org/servlet/opac?punumber=6170933. 978-0-7381-7232-3 (Electronic).

ISO/IEC 9126. 2003. Software engineering - Product quality. 2003.

Jocklin, Jouni. 2011. Koneenohjausjärjestelmän ylläpito. 2011. Master's thesis.

Kääriäinen, Jukka and Välimäki, Antti. 2009. Applying Application Lifecycle Management for the Development of Complex Systems: Experiences from the Automation Industry. Software Process Improvement. s.l. : Springer Berlin Heidelberg, 2009, Vol. 42, pp. 149-160.

Keyes, Jessica. 2004. Software Configuration Management. s.l. : Auerbach Publications, 2004. 0-8493-1976-5.

Lacheiner, Herman and Ramler, Rudolf. 2011. Application Lifecycle Management as Infrastructure for Software Process Improvement and Evolution: Experience and Insights from Industry. Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on. August 30, 2011, pp. 286-293.

Leon, Alexis. 2005. Software Configuration Management Handbook. Second Edition. s.l. : Artech House, 2005. 1-58053-882-7.

Montavista. 2012. MontaVista embedded Linux software and development tools for intelligent devices and embedded systems. [Online] 2012. [Cited: October 4, 2012.] http://www.mvista.com.

Nohau. 2012. Develop Linux Based Embedded Systems. [Course]. Hyvinkää, Finland : s.n., November 21.-23., 2012.

OpenEmbedded. 2012. Main Page - OpenEmbedded.org. [Online] 2012. [Cited: October 5, 2012.] http://www.openembedded.org/wiki/Main_Page.

Poky. 2012. Poky Platform Builder. [Online] November 2, 2012. [Cited: November 2, 2012.] http://www.pokylinux.org/.

Schach, Stephen. 2010. Object-Oriented and Classical Software Engineering. Eigth edition. s.l. : McGraw-Hill, 2010. 0073376183.

Stark, John. 2011. Product Lifecycle Management - 21st Century Paradigm for Product Realisation. s.l. : Springer London, 2011. 978-0-85729-546-0.

Timesys. 2012. Embedded Linux From A Trusted Source | Timesys Embedded Linux. [Online] 2012. [Cited: October 4, 2012.] http://timesys.com.

TUT. 2012. TKT-2431 SoC Design. Department of Computer Systems, Tampere University of Technology. 2012. Course.

TUT. 2011. TKT-3541 SoC Platforms. Department of Computer Systems, Tampere University of Technology. 2011. Course.

Wind River. 2012. Wind River Linux 5. [Online] August 2012. [Cited: November 6, 2012.] http://www.windriver.com/products/product-overviews/Wind-River-Linux-5_Product-Overview.pdf.

Yocto Project. 2012b. The Yocto Project Development Manual. [Online] 2012b. [Cited: October 5, 2012.] http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html.

Yocto Project. 2012a. Yocto Project | Open Source embedded Linux build system, package metadata and SDK generator. [Online] 2012a. [Cited: October 4, 2012.] http://www.yoctoproject.org/.

APPENDIX A: RECIPES

```
DISTRO = "gcc-4.7.2"
DISTRO_NAME = "PCCP with gcc 4.7.2"
DISTRO_VERSION = "1.0+snapshot-${DATE}"

MAINTAINER = "Arttu Leppäkoski <user@mail.com>"

require include/distro-common.inc
```

Code 1: Distro configuration (gcc-4.7.2.conf)

```
IMAGE_INSTALL = “packagegroup-core-boot
                 ${ROOTFS_PKGMANAGE_BOOTSTRAP}
                 ${CORE_IMAGE_EXTRA_INSTALL}”
```

Code 2: Image baseline recipe (product-a.inc)

```
DESCRIPTION = “Remote monitoring application”
SUMMARY = “Application that realize connections to execute
           remote monitoring capabilities”
AUTHOR = “Arttu Leppäkoski <user@mail.com>”
HOMEPAGE = “http://www.company.com/wiki/remotemon”

SECTION = “connectivity”
LICENSE = “MIT”
LIC_FILES_CHKSUM = “file://${COMMON_LICENSE_DIR}/MIT;md5=
                    0835ade698e0bcf8506ecda2f7b4f302”
PR = “r0”
PV = "1.0+svnr${SRCPV}"

PACKAGES = “${PN}”

SRCREV_pn-remotemon ?= "${AUTOREV}"
SRC_URI = “svn://${PCCP_SVN_MIRROR};protocol=http;
           module=remotemon”
S = “${WORKDIR}/remotemon/”

COMPATIBLE_MACHINE = “(pccp_imx35)”

do_install() {
    oe_runmake DESTDIR=${D} install
}

BBCLASSEXTEND = “native”
```

Code 3: Recipe file (remotemon_1.0.bb)

```
DESCRIPTION = “Image Product A 1.3”
SUMMARY = “Release recipe for Product A 1.3”
AUTHOR = “Arttu Leppäkoski <user@mail.com>”
HOMEPAGE = “http://www.company.com/wiki/product-a_1.3”

PR = “r0”
PV = “1.3”

inherit core-image
require product-a

IMAGE_INSTALL_append = “ webserver_1.2”

LICENSE = “MIT”
IMAGE_ROOTFS_SIZE = “8192”

PREFERRED_PROVIDER_virtual/kernel = “linux-pccp”
PREFERRED_VERSION_virtual/kernel = “2.6.31”

PREFERRED_PROVIDER_u-boot = “u-boot-platform”

IMAGE_FEATURES += "ssh-server-dropbear"

ROOTFS_POSTPROCESS_COMMAND += “remove_packaging_data_files”
```

Code 4: Image release recipe (product-a_1.3.bb)

APPENDIX B: YOCTO PROJECT INSTALLATION

Installation steps:
1) Install programs required by the Yocto Project
$ sudo apt-get install sed wget cvs subversion git-core coreutils unzip texi2html texinfo libsdl1.2-dev docbook-utils gawk python-pysqlite2 diffstat help2man make gcc build-essential g++ desktop-file-utils chrpath libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf automake groff cscope makeself

2) Retrieve the Yocto Project and other needed layers and make SVN checkout for the meta-pccp
$ mkdir ~/Yocto
$ cd ~/Yocto
$ git clone -b danny git://git.yoctoproject.org/poky.git
$ cd poky
$ git clone -b danny git://github.com/Freescale/meta-fsl-arm
$ git clone -b danny git://github.com/openembedded/meta-oe.git
$ svn checkout http://company.com/svn/meta-pccp
NOTE: Remember to update and commit this directory regularly to ensure latest changes to be shared between all developers
3) Yocto is now installed to your computer